

érase una vez  
**docker**



Manuel Morejón

# Érase una vez Docker

Crear, compartir y ejecutar aplicaciones modernas

Manuel Morejón

Este libro está a la venta en <http://leanpub.com/erase-una-vez-docker>

Esta versión se publicó en 2022-09-26



Éste es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

© 2021 - 2022 Manuel Morejón

# **¡Tuitea sobre el libro!**

Por favor ayuda a Manuel Morejón hablando sobre el libro en [Twitter](#)!

El tuit sugerido para este libro es:

[¡Todos podemos aprender Docker! #EraseunavezDocker @morejon85](#)

El hashtag sugerido para este libro es [#EraseunavezDocker](#).

Descubre lo que otra gente dice sobre el libro haciendo clic en este enlace para buscar el hashtag en Twitter:

[#EraseunavezDocker](#)

*A mi esposa*

# Índice general

Acerca de este libro . . . . .	1
¿Por qué se ha escrito? . . . . .	1
¿Cómo utilizarlo? . . . . .	1
Conocimientos previos necesarios . . . . .	2
Código fuente . . . . .	2
Comentarios y sugerencias . . . . .	2
Introducción a los contenedores . . . . .	4
Funciona en mi máquina . . . . .	4
¿Qué son los contenedores? . . . . .	7
¿Qué es Docker? . . . . .	9
Los contenedores no son máquinas virtuales . . . . .	12
Resumen del capítulo . . . . .	13
Instalando Docker . . . . .	14
Arquitectura y componentes . . . . .	14
Docker Desktop: definición, precios y suscripciones . . . . .	15
Docker Desktop para Mac . . . . .	17
Docker Desktop para Windows . . . . .	19
Docker Desktop para Linux . . . . .	23
Docker Engine para Linux . . . . .	25
Mi primer contenedor . . . . .	29
Resumen del capítulo . . . . .	30
Primeros pasos con Docker . . . . .	31
Estructura de comandos . . . . .	31
<i>Hola Mundo</i> en detalles . . . . .	34
Registros de imágenes . . . . .	36
Los nombres de las imágenes . . . . .	38
Gestión de imágenes y contenedores . . . . .	40
Cambiando el formato en Docker CLI . . . . .	43
Resumen del capítulo . . . . .	45

## ÍNDICE GENERAL

<b>Rutinas con los contenedores</b> . . . . .	47
Identificar el comando de inicio . . . . .	47
Modificar el comando de inicio . . . . .	50
Iniciar contenedores de forma interactiva . . . . .	53
Iniciar múltiples servicios independientes . . . . .	54
Consultar los logs . . . . .	55
Conocer el consumo de recursos . . . . .	56
Ejercicios para practicar lo aprendido . . . . .	58
Resumen del capítulo . . . . .	58
<b>Publicar y consumir servicios</b> . . . . .	60
Estructura de redes en Docker . . . . .	60
Conocer la IP del contenedor . . . . .	63
Comunicación entre contenedores . . . . .	64
Publicar los puertos en la máquina . . . . .	68
Ejercicios para practicar lo aprendido . . . . .	70
Resumen del capítulo . . . . .	70
<b>Configuraciones y persistencia de datos</b> . . . . .	72
¿Por qué pierdo los datos? . . . . .	72
Persistencia de datos en volúmenes . . . . .	75
Diferentes configuraciones por entorno . . . . .	79
Configuraciones y volúmenes en el mismo contenedor . . . . .	82
Eliminar volúmenes y recuperar espacio . . . . .	83
Ejercicios para practicar lo aprendido . . . . .	85
Resumen del capítulo . . . . .	86
<b>Construcción de imágenes</b> . . . . .	87
Las imágenes se estructuran por capas . . . . .	87
La importancia del fichero Dockerfile . . . . .	90
Crear una imagen desde cero . . . . .	91
Comprendiendo el comando <i>build</i> . . . . .	94
Ejemplo de imagen para un sitio web . . . . .	97
Ejemplo de imagen para aplicación con dependencias . . . . .	99
Ganando velocidad utilizando la caché . . . . .	102
Construir para múltiples plataformas . . . . .	105
Utilizar estructura multi-stage en los Dockerfiles . . . . .	107
Ejercicios para practicar lo aprendido . . . . .	111
Resumen del capítulo . . . . .	111
<b>Publicar imágenes</b> . . . . .	113
Publique su primera imagen en ttl.sh . . . . .	114
Gestionar múltiples etiquetas por imagen . . . . .	117
Publicar imágenes en Docker Hub . . . . .	120

## ÍNDICE GENERAL

Publicar imágenes en GitHub . . . . .	124
Ejercicios para practicar lo aprendido . . . . .	126
Resumen del capítulo . . . . .	127
<b>Desplegar contenedores en la nube . . . . .</b>	<b>128</b>
<b>Docker Compose V2 . . . . .</b>	<b>129</b>
<b>Recomendaciones y Próximos Pasos . . . . .</b>	<b>130</b>

# Acerca de este libro

¡Hola!

¡Bienvenido al fascinante mundo de los contenedores! La industria del software evoluciona de forma rápida hacia arquitecturas nativas de la nube, y sin duda alguna, los contenedores son grandes protagonistas de esta evolución. Los contenidos alrededor de este tema son muchos y muy variados, de ahí la importancia de contar con la documentación precisa y directa que pueda situarle en el camino correcto desde el inicio.

El libro ha sido creado bajo estos principios, teniendo como objetivo principal brindar los conocimientos necesarios para iniciar su viaje en esta tecnología.

El libro está estructurado de menor a mayor grado de complejidad. Los primeros conceptos e instrucciones son básicos, pero luego va a notar el aumento de la complejidad a medida que avanzan los capítulos. Docker es el gran protagonista de este viaje, pero también tendrá la oportunidad de realizar interesantes ejercicios con otras herramientas muy utilizadas en los entornos tecnológicos actuales.

Sin importar el rol que desempeñe en su equipo, desarrollador u operaciones, este libro le brindará los conocimientos necesarios para adentrarse en el entorno de contenedores y en específico, Docker.

Tal vez usted se preguntará, **¿por qué otro libro de Docker? y ¿por qué en Español?**

Es cierto que existen múltiples libros sobre Docker, pero la gran mayoría son en Inglés. Mi criterio sobre el aprendizaje es que los conceptos fundamentales se aprenden y se interiorizan mucho mejor cuando se hace en el idioma nativo, luego siempre hay tiempo para ampliar los conocimientos en el idioma que deseé la persona.

## ¿Por qué se ha escrito?

Las razones son las siguientes:

- Apoyar y aumentar la documentación existente sobre Docker en Español.
- Brindar una guía de aprendizaje para los que se inician en el fascinante mundo de Docker.
- Compartir soluciones para resolver los retos encontrados al trabajar con contenedores.
- Brindar sugerencias cuando se necesita tomar decisiones en el trabajo con contenedores.

## ¿Cómo utilizarlo?

Si desea llevar a la práctica rápidamente los conocimientos aprendidos sobre Docker, le recomiendo sentarse delante de su ordenador, descargar la versión digital del libro y abrir su Terminal preferido.

Podrá leer las secciones de su interés e ir poniendo en práctica los conceptos aprendidos sin necesidad de costo alguno en infraestructura.

Si solamente desea revisar algunos conceptos sobre los contenedores, puede descargar el libro en el formato que desee y acceder a las secciones que le sean de interés. En los capítulos va a encontrar enlaces a sistemas y herramientas que le permitirán aumentar los conocimientos en profundidad.

## Conocimientos previos necesarios

No se necesita tener conocimientos previos sobre [Docker](#)<sup>1</sup> para sacar provecho del libro.

Se recomienda tener conocimientos básicos en administración de sistemas Linux.

## Código fuente

Las configuraciones y ejemplos utilizados en el libro se encuentran GitHub.

### Repositorios

- <https://github.com/mmorejon/erase-una-vez-docker>
- <https://github.com/mmorejon/erase-una-vez-1>
- <https://github.com/mmorejon/erase-una-vez-2>

## Usuarios en Windows

Se recomienda utilizar [GitBash](#)<sup>2</sup> en el momento de ejecutar los comandos en consola. Este sistema le va a brindar un entorno de consola similar a Linux.

## Comentarios y sugerencias

Su criterio es el más importante. En todo momento estaré al tanto de sus dudas e inquietudes para ayudarlo a encontrar respuestas.

Puede escribirme a través de los siguientes canales para compartir sus sugerencias y comentarios.

- Correo desde el formulario de Leanpub<sup>3</sup>
- Mensaje directo en Twitter<sup>4</sup>

---

<sup>1</sup><https://www.docker.com>

<sup>2</sup><https://gitforwindows.org/>

<sup>3</sup>[https://leanpub.com/erase-una-vez-docker/email\\_author/new](https://leanpub.com/erase-una-vez-docker/email_author/new)

<sup>4</sup><https://twitter.com/morejon85>

- Mensaje directo en LinkedIn<sup>5</sup>

Después de haber avanzado en la lectura de los capítulos, si considera que ha sido útil el contenido, vendría genial una revisión en la plataforma donde ha comprado el libro para que sirva de referencia al resto de lectores.

El libro todavía se encuentra en fase de desarrollo, por lo tanto, si desea sugerir capítulos o modificaciones en secciones hágallo saber y con gusto será analizado.

¡Muchas gracias por adelantado!

---

<sup>5</sup><https://www.linkedin.com/in/manuelmorejon/>

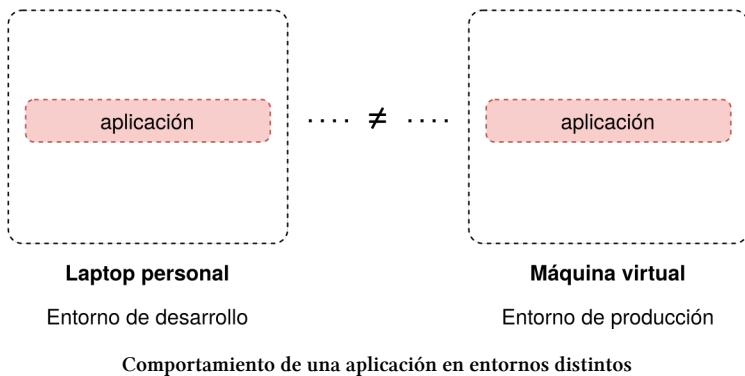
# Introducción a los contenedores

En el capítulo conocerá los problemas que dieron origen a los contenedores. También aprenderá el significado de Docker como herramienta, comunidad y ecosistema. Esta será la base del conocimiento que va a necesitar para dar los primeros pasos en el mundo de los contenedores y en específico Docker.

## Funciona en mi máquina

Seguramente la frase “*el código funciona en mi máquina, pero no en el servidor*” le resulte familiar. Esta situación ha sucedido durante mucho tiempo en los equipos de desarrollo, de hecho, puede ser que usted vea este problema con frecuencia.

Este problema tiene lugar cuando se pone en ejecución el código de la aplicación en un entorno diferente al lugar donde se ha creado, por ejemplo, cuando el código pasa de la máquina del desarrollador a los entornos de calidad o producción.

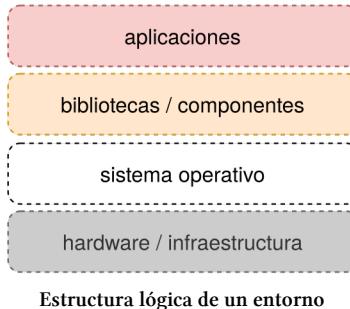


La clave a tener en cuenta en esta situación son los entornos, de aquí lo importante de hacer la siguiente pregunta: *¿cuántas diferencias existen entre estos entornos?*

Para responder esta pregunta lo primero será definir, *¿qué es un entorno para nosotros?* Un *entorno* es el lugar donde se ejecuta el código programado. Un entorno de producción puede ser la máquina virtual creada en alguna de las conocidas plataformas públicas como AWS, GCloud, Azure, u otra similar. El entorno de pruebas también puede estar compuesto por máquinas virtuales en plataformas públicas, mientras que el entorno de desarrollo es el ordenador utilizado para programar el código (*nuestra máquina*).

Cada uno de los entornos mencionados está compuesto por las siguientes capas o estructuras lógicas: hardware/infraestructura, sistema operativo, bibliotecas/componentes y aplicaciones; siendo esta

última (aplicaciones) el lugar donde se encuentra el código en ejecución. Tomando como referencia estas capas puede identificar de forma rápida muchas de las diferencias que existen entre los entornos de trabajo, principalmente entre los entornos de desarrollo y producción. Los entornos de desarrollo (*nuestras máquinas*) tienen sistemas operativos con interfaz gráfica que apoya a un rápido desarrollo de software, mientras que los servidores de producción utilizan distribuciones orientadas a sacar el máximo rendimiento de los recursos, donde el acceso se realiza principalmente a través de la interfaz de comandos.



Estructura lógica de un entorno

Grandes diferencias pueden existir también entre las bibliotecas instaladas en ambas máquinas. Dependiendo del tipo de aplicación necesita instalar componentes asociados al lenguaje de programación utilizado. Para el lenguaje Java se necesita Java Development Kit (JDK), en Golang el binario de la versión en uso, y así sucesivamente con cada uno de los posibles lenguajes. Estará de acuerdo conmigo que las versiones instaladas en las máquinas de desarrollo casi nunca son las mismas a las instaladas en los servidores de producción. Entre las causas más repetidas están:

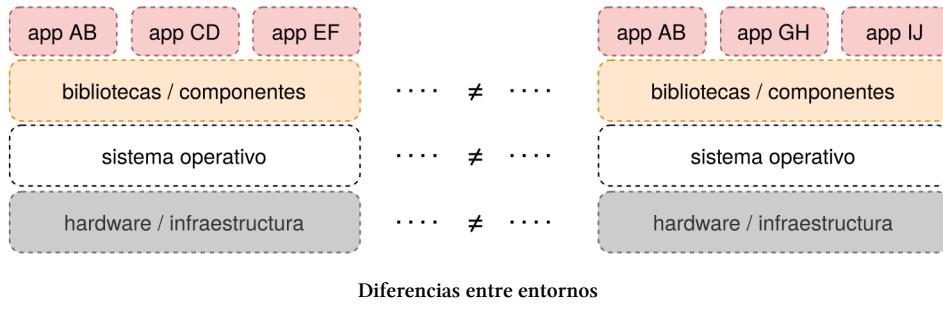
- Existen múltiples aplicaciones funcionando en el mismo servidor con el mismo lenguaje de programación, pero con diferentes versiones.
- En el entorno de desarrollo se tiene instalada la última versión del lenguaje de programación para hacer pruebas de las nuevas funcionalidades, pero en el servidor de producción debe ser instalada una versión estable por requerimientos de seguridad.

Aunque estos elementos deben ser suficientes para entender las diferencias entre los entornos, agreguemos un elemento más, *las configuraciones*. Las configuraciones y parámetros establecidos en los entornos de producción son completamente distintas a los entornos de desarrollo. En este punto no hay forma de lograr que sean iguales. Tanto por requerimientos de seguridad, como por requisitos de la plataforma, las configuraciones siempre son distintas.

Llegados a este punto, donde quedan claras las diferencias entre entornos, volvamos a hacer la pregunta:

*¿Por qué el código funciona bien en mi máquina, pero no en el servidor de producción?*

Seguramente se imagina la respuesta, porque los entornos son distintos.



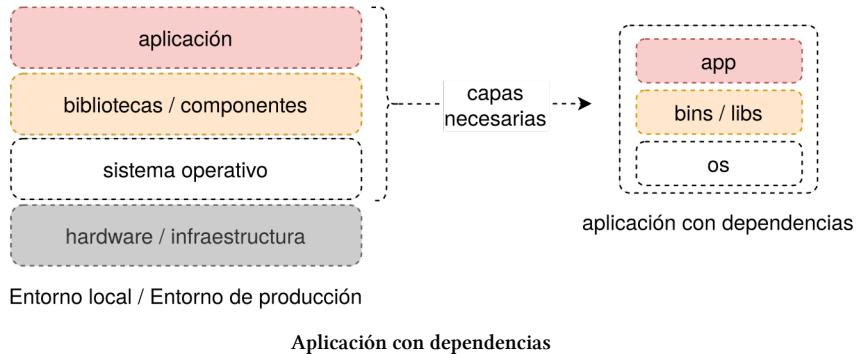
## Dependencias y componentes de una aplicación

Para garantizar el mismo funcionamiento de las aplicaciones en diferentes entornos hay que tener en cuenta todas sus dependencias y componentes. Para una aplicación web sencilla se pueden obtener las siguientes dependencias:

- código de la aplicación (*html, css, javascript, imágenes*)
- servidor web donde se ejecuta la aplicación (*nginx o apache*)
- bibliotecas y dependencias del lenguaje de programación
- sistema operativo

Si se desea que este ejemplo funcione bien en cualquier entorno, será necesario reproducir exactamente los cuatro puntos en el lugar donde se ejecute. Sería muy sencillo si se pudiese copiar/pegar los componentes como si fueran ficheros en el ordenador, ¿cierto?

De las capas mencionadas, el único elemento no obligatorio es el hardware/infraestructura, porque no es posible mover su ordenador hacia Amazon, pero para el resto de capas será interesante pensar cómo ponerlas en un solo paquete y moverlas entre entornos.



¿Será posible empaquetar la aplicación con todas sus dependencias?

Sí es posible, se puede lograr utilizando **Contenedores**.

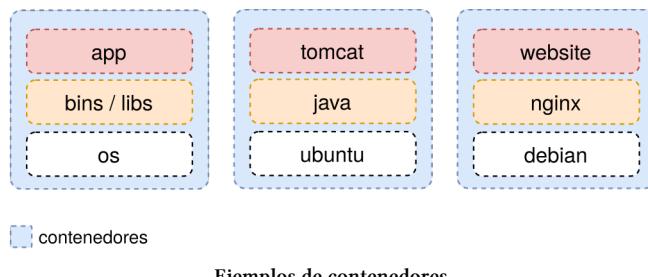
## ¿Qué son los contenedores?

Un contenedor es una unidad de *software* estandarizada que contiene todas las dependencias necesarias para su ejecución en cualquier entorno<sup>6</sup>. Un contenedor es un proceso en ejecución.

Las principales características de los contenedores son:

- **Ligeros:** En un contenedor se incluyen solamente los ficheros que son necesarios para su funcionamiento.
- **Portables:** Pueden ser empaquetados en un fichero y ser distribuidos de forma rápida y sencilla hacia los entornos. Cuando llega a su destino no necesita dependencia adicionales para su ejecución.
- **Procesos aislados:** Los procesos dentro de un contenedor se encuentran aislados del resto de procesos de la máquina.

A continuación se muestran ejemplos de contenedores presentes en la industria.



Estos ejemplos representan una pequeña muestra de los contenedores que verá en los próximos capítulos. Es importante mencionar que todos cumplen con las características mencionadas anteriormente. Al ver la zona del sistema operativo se preguntará que tan ligero puede llegar a ser un Ubuntu Server. El fichero original cuenta con 1.2GB, mientras que en el contenedor solamente son 73MB (*en el momento de escribir el libro*). La diferencia se debe a que han sido eliminados de todos aquellos elementos no son necesarios para ejecutar la aplicación.

## Una imagen no es un contenedor

Una palabra muy cercana al concepto contenedor es *imagen*. En muchos lugares se hace referencia al concepto de *imagen* de forma incorrecta, de aquí la importancia de dejar explicados ambos conceptos lo antes posible.

Una *imagen* es una unidad de *software* estandarizada compuesta por múltiples ficheros. Esta definición es bien similar a la utilizada para definir los contenedores, pero la diferencia radica en

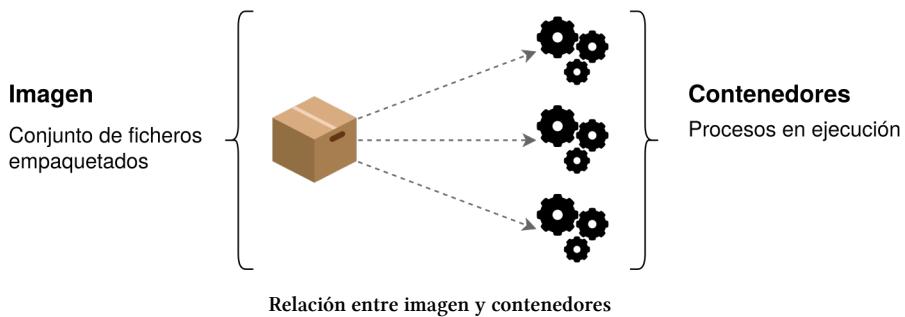
<sup>6</sup><https://www.docker.com/resources/what-container>

la ausencia de un estado. Las imágenes no pueden estar iniciadas, detenidas o pausadas porque no son procesos, solamente son ficheros.

Las imágenes están en nuestro ordenador como cualquier otro fichero, pueden ser movidas de una máquina a otra, o transferidas hacia un almacenamiento en la nube, pero nunca en ejecución.

Sin embargo, los contenedores sí son procesos en ejecución, y siempre tienen un estado asociado.

Existe una estrecha relación entre las imágenes y los contenedores, con una imagen pueden ser iniciados muchos contenedores. Si se busca un homólogo con elementos de cocina se puede decir que la imagen es la receta y los contenedores son los platos que se hicieron siguiendo la receta. Otro ejemplo, pero más cercano al mundo de la programación, puede ser que una imagen es una clase y los contenedores son los objetos. De una clase pueden crearse muchos objetos. En resumen, la imagen es el conjunto de ficheros (*sistema operativo, dependencias y código de la aplicación*) que utiliza el contenedor para iniciarse como un proceso.



En la siguiente tabla se muestran las principales diferencias entre las imágenes y los contenedores.

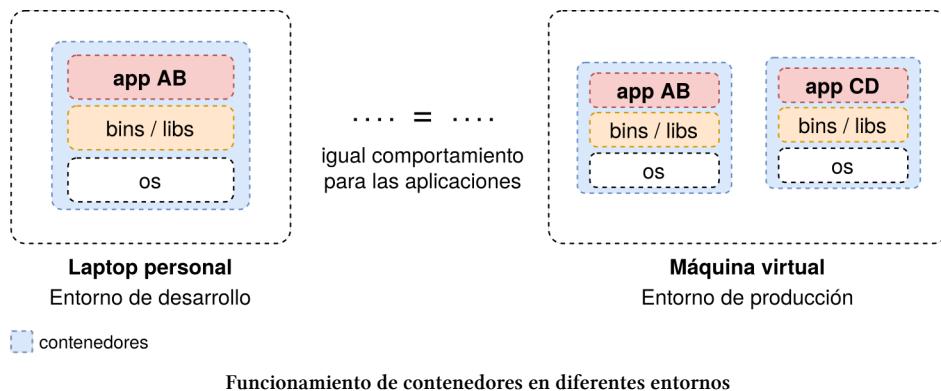
Imagen	Contenedor
conjunto de ficheros empaquetados	es un proceso en ejecución
no tiene estado	tiene estado, p.ej iniciado, detenido, pausado
principales comandos: build, push y pull	principales comandos: run, stop, exec

## Queda resuelto el problema de las dependencias

Ahora que conoce el significado de *Contenedor*, se puede volver al problema planteado inicialmente, donde el sistema desarrollado en el entorno local no muestra el mismo comportamiento en el entorno de producción.

Con los contenedores el problema queda resuelto, porque dentro de los contenedores están incluidas las dependencias necesarias para que el programa se ejecute de igual forma en todo momento.

La siguiente imagen muestra la puesta en ejecución del sistema en ambos entornos utilizando contenedores.



Se nota fácilmente que cada aplicación tiene sus dependencias empaquetadas dentro del contenedor, por lo tanto, su funcionamiento va a ser el mismo. Un segundo aspecto que resalta en la imagen es el aislamiento entre aplicaciones, donde cada una es un contenedor independiente. Entre contenedores no se comparten los procesos.

El tener resuelto el problema inicial es solamente el principio, todavía falta por responder nuevas incógnitas, como por ejemplo:

*¿Cuál herramienta permite gestionar las aplicaciones a través de contenedores?*

Existen múltiples herramientas para gestionar las aplicaciones como contenedores, pero hay una que ha sido capaz de revolucionar la industria del software, **Docker**.

## ¿Qué es Docker?

Docker<sup>7</sup> es la plataforma por excelencia para gestionar contenedores, y es a su vez, una herramienta. El principal objetivo de Docker es *construir, distribuir y ejecutar* aplicaciones en contenedores de forma fácil y sencilla.



Logo de Docker

Los conceptos sobre contenedores han estado presentes en la industria desde hace mucho tiempo, pero en todos los casos ha existido gran complejidad para su uso y comprensión. De igual forma la industria carecía de un ecosistema apropiado para extender esta tecnología a gran escala. Esta fue la gran virtud de Docker, facilitar el uso de los contenedores en todos los entornos y crear un ecosistema de herramientas para su uso.

<sup>7</sup><https://www.docker.com/>

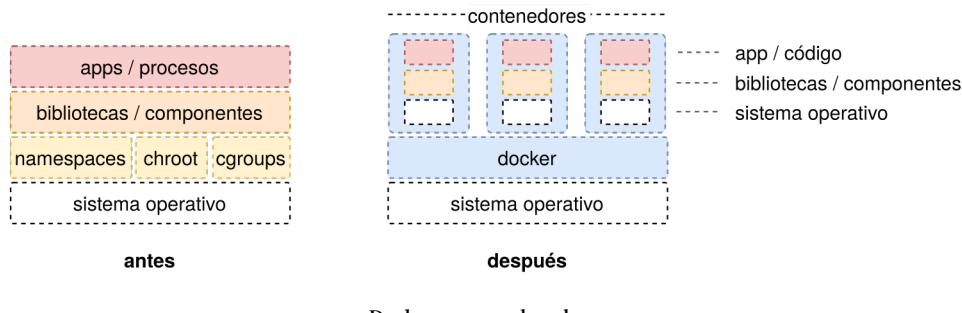
## ¿Cuál es el significado de la palabra **docker**?

La palabra *docker* significa *estibador*. Un estibador es la persona que está en el puerto ayudando a bajar y subir mercancía de los barcos. En los puertos la mercancía viene en contenedores, por lo tanto, el nombre a la herramienta va genial.

Se puede buscar rápidamente la analogía entre un puerto de barcos y el desarrollo de aplicaciones. Las aplicaciones se desarrollan en los ordenadores, luego son empaquetadas, se mueven a los entornos en la nube y por último se despliegan en entornos de producción. Pues con los barcos es similar, todas las mercancías deben ser empaquetadas y luego subidas al barco para distribuirlas hacia su destino final.

## ¿Cómo es Docker en un ordenador?

Docker utiliza componentes del Kernel de Linux para garantizar el aislamiento de los procesos (los contenedores). Los principales componentes utilizados por Docker son los *namespaces*, *chroot* y *cgroups*.



Docker en un ordenador

Docker se instala en el ordenador como cualquier otro programa. Una vez iniciado es un proceso más en la máquina, pero con la responsabilidad de gestionar contenedores. El número de contenedores que a gestionar en su ordenador va a ser proporcional a los recursos de CPU y RAM que posea el equipo.

## ¿Quién utiliza Docker?

Docker es utilizado por desarrolladores y administradores de sistemas, ambos grupos se ven beneficiados al contar con los siguientes elementos:

- Se reduce el tiempo de instalación y configuración de las aplicaciones y servicios.
- Se evitan errores de compatibilidad al utilizar múltiples versiones de una aplicación.
- Rápido empaquetado y despliegue de las aplicaciones hacia entornos en la nube.
- Permite escalar las aplicaciones en tiempo real con gran facilidad.
- Garantiza el correcto funcionamiento en múltiples entornos.

Estos son algunos de los múltiples motivos que convierten a Docker la herramienta más popular para gestionar contenedores, pero este es solo el comienzo todavía quedan más sorpresas.

## Ecosistema creado por Docker

Además de estar Docker como herramienta, existe un conjunto de sistemas y aplicaciones que complementan el trabajo con los contenedores. Este conjunto de elementos que colaboran entre sí se conoce como ecosistema, y entre los principales sistemas se encuentran:

### Hub

Docker Hub<sup>8</sup> es la plataforma donde se almacenan las imágenes de los contenedores. En ella puede crearse su usuario personal, corporativo o ambas. El objetivo de *Docker Hub* es brindar un registro centralizado para todas sus aplicaciones.

Este patrón existía en la industria para distribuir la información. Casi todas las tecnologías cuentan con un almacenamiento para sus artefactos digitales, p.ej registros para node, python, ruby, entre otros muchos ejemplos.

*Docker Hub* es el principal almacenamiento de imágenes públicas en la industria. A través de su buscador puede encontrar imágenes oficiales de todas las tecnologías, desde lenguajes de programación, sistemas de bases de datos, monitorización y cualquier otra que pueda imaginarse.

Otro detalle importante es la privacidad de la información, si usted lo desea puede almacenar las imágenes de forma privada, garantizando su acceso solamente a aquellas personas o grupos de su interés.

A medida que avance en el libro recibirá indicaciones de cómo trabajar con esta plataforma y aprenderá a sacar provecho de ella.

### Compose

Docker Compose<sup>9</sup> permite gestionar múltiples contenedores al mismo tiempo. Es un orquestador de contenedores fácil e intuitivo de utilizar.

Durante el desarrollo de los microservicios es común hacer uso de bases de datos o tal vez interactuar con otros microservicios, es aquí donde *Docker Compose* hace gala de sus bondades, permitiendo gestionar estas integraciones a través de un fichero de configuración.

### Registry

Docker Registry<sup>10</sup> permite crear registros privados de imágenes de contenedores. Puede instalar este sistema en su plataforma y administrar directamente el almacenamiento de sus imágenes. Este sistema le brinda el control total del registro de imágenes.

---

<sup>8</sup><https://hub.docker.com/>

<sup>9</sup><https://docs.docker.com/compose/>

<sup>10</sup><https://docs.docker.com/registry/>

## Trust

Docker Trust<sup>11</sup> tiene la responsabilidad de firmar digitalmente las imágenes de contenedores durante el proceso de construcción. Esta funcionalidad brinda la garantía de saber que está utilizando su imagen y que no ha sido interceptada o cambiada durante el proceso de distribución.

Recuerde que la seguridad de sus aplicaciones en la cadena de distribución es un tema muy importante.

## Los contenedores no son máquinas virtuales

Hasta el momento se han comentado las bondades de los contenedores: son ligeros, portables y contienen todas las dependencias de una aplicación. Sin embargo, estas características se parecen mucho a los beneficios que brindan las máquinas virtuales, *¿será entonces que los contenedores son máquinas virtuales ligeras?*

La respuesta es no, los contenedores no son máquinas virtuales.

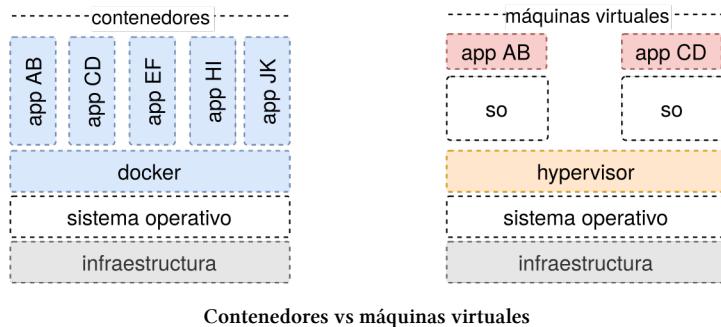
Es importante resaltar esta respuesta para no dar lugar a equivocaciones. El hecho de existir elementos comunes entre ambas tecnologías no significa que sean iguales. Las principales diferencias entre ambas se encuentran en los siguientes puntos:

- *Aislamiento*: Las máquinas virtuales son sistemas operativos funcionando completamente aislados dentro de la máquina local. Para crearlas se utilizan herramientas de virtualización p.ej [VirtualBox<sup>12</sup>](https://www.virtualbox.org/). Por otro lado, los contenedores son procesos ligeros funcionando de forma aislada. Estos utilizan los recursos del Kernel de Linux. Los contenedores no dependen de las herramientas de virtualización.
- *Espacio en disco*: El tamaño de las máquinas virtuales se mide en *gigabytes* (GB), son estructuras bastante pesadas porque necesitan de todo el sistema operativo para su funcionamiento. El tamaño de los contenedores es mucho menor y suele medirse en *megabytes* (MB). Se comparten muchos componentes del Kernel de Linux, por lo que el contenedor solamente adiciona los ficheros de la aplicación en ejecución. Existen contenedores con apenas 5 MB.
- *Construcción*: Las máquinas virtuales se crean a partir de la instalación y puesta en marcha de un sistema operativo para resolver múltiples propósitos. En esta máquina virtual pueden ejecutarse muchos servicios al mismo tiempo y cubrir diversas necesidades. Sin embargo, en los contenedores es totalmente a la inversa. Los contenedores se construyen con los ficheros necesarios para ejecutar la aplicación. Un contenedor debe resolver solamente una necesidad.
- *Velocidad de inicio*: Un contenedor es mucho más rápido que una máquina virtual a la hora de iniciarse, detenerse o reiniciarse. La diferencia en la velocidad es una consecuencia directa de las diferencias explicadas anteriormente.

Debido a estas diferencias la industria ha adoptado y extendido el uso de contenedores a través de todas las plataformas.

<sup>11</sup><https://docs.docker.com/engine/security/trust/>

<sup>12</sup><https://www.virtualbox.org/>



## Resumen del capítulo

En el capítulo se analizó por qué muchas de las aplicaciones no funcionan de la forma esperada cuando pasan del entorno de desarrollo al entorno de producción. La solución es utilizar los contenedores para empaquetar las aplicaciones junto con sus dependencias. De esta forma se puede garantizar el correcto funcionamiento de la aplicación sin importar el entorno donde se encuentre.

Luego fue presentado Docker, herramienta pensada para construir, distribuir y ejecutar contenedores de forma rápida y sencilla. Docker ha modificado la industria del software en todas sus aristas, de ahí la importancia de conocer y dominar esta herramienta.

Se ha explicado que Docker no es solamente una herramienta, sino que también es un ecosistema. Múltiples aplicaciones colaboran entre sí para brindar la mejor experiencia al trabajar con contenedores. Las principales aplicaciones fueron mencionadas de forma rápida, pero a medida que avancen los capítulos serán analizadas nuevas herramientas.

Este es solamente el inicio, al haber presentado Docker se abre un sin fin de nuevos conocimientos a explorar. *¡Siga con nosotros para que también pueda sacar provecho de ellos!*

# Instalando Docker

Instalar Docker es muy sencillo, con pocos clics en su ordenador, o con un par de comandos en su Terminal, puede hacer uso de los contenedores. Sin embargo, existen múltiples sistemas operativos, y en cada uno existen detalles que deberá conocer para sacar el máximo de provecho de la herramienta. De aquí que el objetivo del capítulo sea mostrar el camino a seguir ante tanta diversidad de sistemas operativos.

La primera sección va a corresponder con la arquitectura de Docker como herramienta. Siempre es importante conocer cómo está estructurado un sistema para poder utilizarlo correctamente.

Luego se explican las diferentes suscripciones que ofrece Docker como empresa. Conocer las diferencias entre los planes existentes le van ayudar a identificar sus necesidades. Le puedo adelantar que ***no es necesario realizar ningún pago en su condición de estudiante***, pero existen casos donde las empresas grandes deben asumir un coste adicional por utilizar *Docker Desktop*.

Hasta el momento no se ha explicado qué es *Docker Desktop* o *Docker Engine*, pero ambos elementos forman parte de las secciones que vienen a continuación. Al terminar el capítulo va saber en qué momento utilizar cada una según sea el caso.

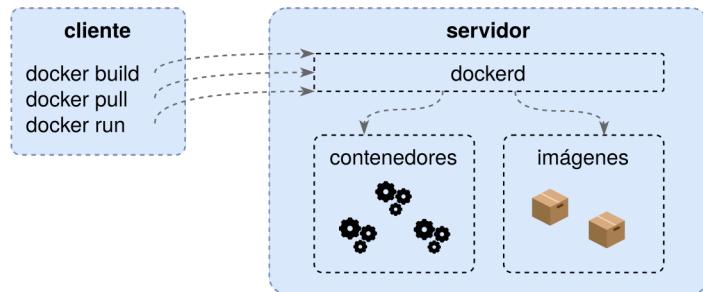
Después de las secciones de instalación se va a realizar la primera actividad con contenedores *¡Hola Mundo!*. Toda tecnología tiene un primer ejemplo fácil y rápido de ejecutar, y por su puesto, los contenedores también tienen el suyo.

¡Tenga listo su ordenador, que ya empezamos!

## Arquitectura y componentes

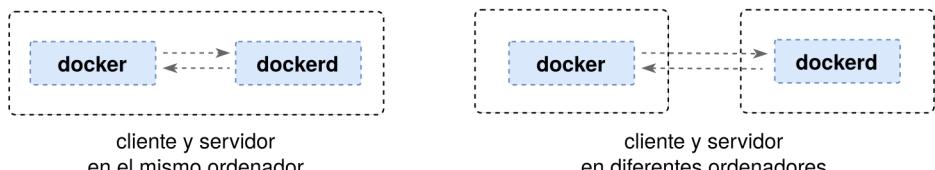
Si conoce la arquitectura de un sistema podrá obtener el máximo provecho de sus funcionalidades.

La arquitectura de Docker es sencilla, pero robusta. Utiliza un modelo *cliente - servidor*, donde el cliente brinda la interfaz de comandos (*docker*), y el servidor (*dockerd*) es el proceso que realiza las operaciones en la máquina.



Vista de la arquitectura

Ambos elementos pueden estar instalados en la misma máquina, que de hecho es lo que va a suceder cuando llegue a la sección de instalación, pero también puede conectar un cliente local con un servidor *dockerd* en una máquina remota. Entre el cliente y el servidor la comunicación se realiza utilizando API REST, ya sea a través de sockets UNIX o de interfaces de redes.



Cliente y servidor de Docker

Al conjunto *cliente - servidor* se le hace referencia con el nombre [Docker Engine](#)<sup>13</sup>.

## Docker Desktop: definición, precios y suscripciones

### ¿Qué es Docker Desktop?

[Docker Desktop](#)<sup>14</sup> es una aplicación de instalación rápida creada para desarrolladores. Le va a permitir construir y compartir de forma fácil sus aplicaciones y microservicios a través de contenedores. Al instalar Docker Desktop está incluyendo también los siguientes sistemas: Docker Engine, Docker Compose, Docker Content Trust, Kubernetes, Docker Scan y Credential Helper. Estas herramientas le serán muy útiles durante el proceso de desarrollo y podrán ser utilizadas todas desde la misma aplicación Docker Desktop.

Docker Desktop también permite incluir nuevas funcionalidad a través de las Extensiones ([Extensions](#)<sup>15</sup>). Las funcionalidades pueden estar desarrolladas por la comunidad, pero también puede crear

<sup>13</sup><https://docs.docker.com/engine/>

<sup>14</sup><https://docs.docker.com/desktop/>

<sup>15</sup><https://docs.docker.com/desktop/extensions/>

la suya propia basado en sus necesidades. El uso de Extensiones en Docker Desktop abre las puertas a todos los equipos para compartir ideas y herramientas basadas en contenedores.

Docker Desktop se encuentra disponible para Mac, Windows y Linux, pudiendo ser utilizado sin restricciones por casi la totalidad de usuarios. La excepción aparece solamente para las grandes empresas, donde deben pagar por su uso. Desde el punto de vista de Docker, una gran empresa es aquella que tiene más de 250 empleados y recibe más de \$10 millones de ingresos anuales. **Solamente las empresas que cumplan este criterio están obligadas a utilizar una suscripción de pago para hacer uso de la aplicación Docker Desktop.** Dicho de otra manera, si la empresa donde usted trabaja tiene estas características, debe pagarle a usted una suscripción Pro, Team o Business si quiere que usted utilice Docker Desktop. En este momento usted es un estudiante, por lo tanto no tiene que preocuparse por las suscripciones durante el uso del libro.

Si desea conocer los detalles de las políticas aplicadas a las grandes empresas consulte [esta página<sup>16</sup>](#) en el sitio de Docker.

Para entornos gráficos la recomendación es utilizar Docker Desktop, mientras que para servidores y entornos sin interfaz gráfica la recomendación es utilizar directamente [Docker Engine](#). En las próximas secciones se va a mostrar cómo instalar Docker Desktop en Windows, Mac y Linux, así como también Docker Engine en entornos Linux.

## Precios y suscripciones

Es fundamental conocer los términos y condiciones de cada herramienta antes de instalarla. Recuerde que en temas legales, el desconocimiento de las reglas no le exime de culpas. Es su responsabilidad leer la licencia asociada al producto antes de hacer uso del mismo.

Docker tiene sus propios términos y condiciones, de ahí la importancia de prestar atención a esta sección antes de seguir con la instalación del producto. **La vigencia del contenido asociado a términos y condiciones corresponde con el momento en que se ha escrito el libro.**

En este momento Docker cuenta con cuatro tipos de suscripciones:

- **Personal:** Precio \$0 (sin costes). Ideal para desarrolladores independientes, educación, comunidades *open source* y negocios pequeños.
- **Pro:** Precio \$5/mes/por persona. Incluye herramientas profesionales para desarrolladores independientes que deseen acelerar su productividad.
- **Team:** Precio \$7/mes/por persona. Ideal para equipos, incluye elementos para mejorar la colaboración, la seguridad y la productividad.
- **Business:** Precio \$21/mes/por persona. Ideal para medianas y grandes empresas donde se necesita centralizar la gestión de procesos junto con una seguridad avanzada.

De las cuatro suscripciones, la opción que se ajusta a su caso de uso es la primera, donde **no hay que realizar ningún pago**, porque son actividades educativas como desarrolladores independientes. La suscripción utilizada va a ser **Personal**.

---

<sup>16</sup><https://www.docker.com/pricing/faq>

Otro detalle a tener en cuenta, *las suscripciones están relacionadas únicamente con el uso de Docker Desktop*, aplicación de interfaz gráfica desarrollada por Docker. El uso de Docker Engine<sup>17</sup> no tiene ningún tipo de suscripción asociada, puede utilizarlo en cualquier circunstancia. Si desea ampliar la información sobre los precios de las suscripciones utilice este enlace<sup>18</sup>.

## Docker Desktop para Mac

La instalación de Docker para Mac<sup>19</sup> se realiza de forma simple y rápida gracias al uso de Docker Desktop.

### Requisitos previos necesarios

Antes de instalar Docker Desktop para Mac garantice que su ordenador cumple con los requisitos necesarios. [Enlace del sitio oficial<sup>20</sup>](#).

#### Para una Mac con Intel Chip

- Versión de macOS 10.14 o superior. Siempre se recomienda utilizar la última versión.
- Mínimo 4 GB de RAM.
- Si tiene VirtualBox instalado, la versión no puede ser menor a la 4.3.30, porque sería incompatible con Docker.

#### Para una Mac con Apple Silicon M1

- Tener instalado Rosetta 2 porque algunos componentes todavía están en Darwin/AMD64. Para instalar Rosetta 2 utilice el siguiente comando:

```
$ softwareupdate --install-rosetta
```

## Instalación y puesta en marcha

Descargue el fichero Docker Desktop según el chip de su ordenador: [Intel Chip<sup>21</sup>](#) o [Apple Silicon M1<sup>22</sup>](#).

Inicie la instalación dando clic sobre el fichero descargado, luego mueva la aplicación junto con el resto de sistemas de su ordenador.

---

<sup>17</sup><https://docs.docker.com/engine/>

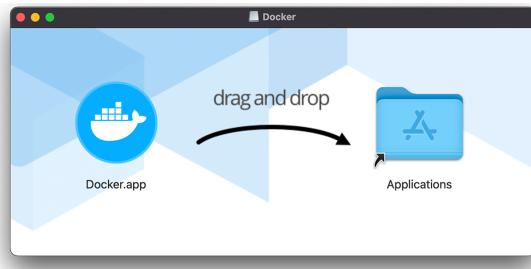
<sup>18</sup><https://www.docker.com/pricing>

<sup>19</sup><https://docs.docker.com/desktop/mac/install/>

<sup>20</sup><https://docs.docker.com/desktop/mac/install/>

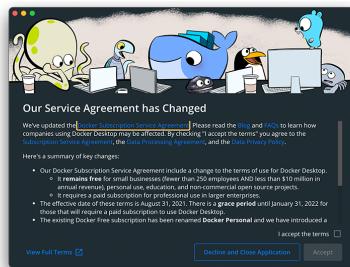
<sup>21</sup><https://desktop.docker.com/mac/main/amd64/Docker.dmg>

<sup>22</sup><https://desktop.docker.com/mac/main/arm64/Docker.dmg>



### Instalando Docker en Mac

Busque en el listado general de aplicaciones el nombre Docker y haga clic sobre ella para iniciarla. Necesita aceptar los términos y condiciones para seguir adelante con la instalación, en este caso son bastante cortos, así que le recomiendo leerlos y continuar si está de acuerdo. Recuerde que en su caso puede utilizar Docker Desktop completamente gratis porque está como estudiante. Este mensaje debe tenerse en cuenta si utiliza Docker Desktop en el ordenador de su empresa.



### Docker términos y condiciones

Con estos pasos ha quedado instalado Docker en su ordenador junto con otros componentes que serán de gran ayuda. En la barra de estado ha sido incluido el ícono de Docker que se utiliza para acceder de forma rápida a las configuraciones.

Al iniciar el sistema aparece el *Dashboard* de la aplicación, puede que aparezcan múltiples mensajes de recomendaciones sobre cómo utilizarla, si desea hacerlo no hay problema, tómese su tiempo explorando las opciones que desee y luego regrese para hacer las primeras pruebas en el Terminal.

El primer comando a utilizar en el Terminal será para conocer la versión de Docker.

```
$ docker version

Client:
Cloud integration: v1.0.25
Version:          20.10.16
API version:      1.41
Go version:       go1.17.10
Git commit:       aa7e414
Built:            Thu May 12 09:20:34 2022
OS/Arch:          darwin/amd64
Context:          default
Experimental:    true

Server: Docker Desktop 4.9.1 (81317)
Engine:
  Version:          20.10.16
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.17.10
  Git commit:       f756502
  Built:            Thu May 12 09:15:42 2022
  OS/Arch:          linux/amd64
  Experimental:    false
  containerd:
    Version:        1.6.4
    GitCommit:      212e8b6fa2f44b9c21b2798135fc6fb7c53efc16
  runc:
    Version:        1.1.1
    GitCommit:      v1.1.1-0-g52de29d
  docker-init:
    Version:        0.19.0
    GitCommit:      de40ad0
```

No pase por alto el detalle que Docker le brinda la versión de ambos componentes, el cliente y el servidor. Ambos elementos fueron explicados en la sección [Arquitectura y Componentes](#).

Si no desea conocer los procesos de instalación para otras distribuciones puede saltar directamente a la sección [Mi primer contenedor](#).

## Docker Desktop para Windows

La instalación de [Docker para Windows](#)<sup>23</sup> se realiza de forma simple gracias a Docker Desktop.

---

<sup>23</sup><https://docs.docker.com/desktop/windows/install/>

## ¿Qué configuración elegir? WSL2 o Hyper-V

Docker necesita que el sistema operativo Windows sea configurado para realizar la gestión de imágenes y contenedores. Esta configuración previa a la instalación puede realizarse de dos formas: a través de *Windows Subsystem for Linux (WSL) 2*<sup>24</sup> o utilizando virtualización *Hyper-V*<sup>25</sup>.

Ambas opciones funcionan correctamente, pero existen diferencias entre ellas que hacen inclinar la balanza hacia el lado de WSL 2.

- Es un *kernel* de Linux que permite la ejecución de contenedores sin necesidad de emulación.
- Consume menos recursos porque utiliza la función de asignación de memoria dinámica presente en WSL 2.
- Mayor velocidad en la construcción de imágenes.
- Mayor velocidad en el inicio del sistema Docker Desktop.
- Mejor integración con volúmenes atados al sistema de archivos.

Estos cinco elementos son un resumen rápido de las principales ventajas de utilizar WSL 2 a la hora de instalar Docker en Windows. De todas formas, ampliar los conocimientos nunca está de más, si desea leer más al respecto le recomiendo consultar [este artículo<sup>26</sup>](#) en el blog oficial de Docker.

Tomando como base este análisis, la instalación a realizar en el libro será la descrita para WSL2 con la distribución Ubuntu. Si de todas formas desea instalar la variante con Hyper-V, puede hacerlo a través de [este enlace<sup>27</sup>](#).

## Requisitos previos necesarios

Los siguientes puntos son obligatorios antes de instalar Docker Desktop en Windows con WSL 2.

- Utilizar Windows 10 versión igual o mayor a 1903 o Windows 11.
- Habilitar WSL 2 en Windows. Puede utilizar el enlace oficial de [Microsoft<sup>28</sup>](#).
- Descargar e instalar la actualización del [kernel de Linux para Windows<sup>29</sup>](#).

## Instalación y puesta en marcha

Descargue [Docker Desktop para Windows<sup>30</sup>](#) en una versión igual o mayor a 2.3.0.2. Haga doble clic en el fichero descargado y siga las instrucciones proporcionadas por el instalador.

<sup>24</sup><https://docs.microsoft.com/es-es/windows/wsl/about>

<sup>25</sup><https://docs.microsoft.com/es-es/windows-server/virtualization/hyper-v/hyper-v-technology-overview>

<sup>26</sup><https://www.docker.com/blog/docker-hearts-wsl-2/>

<sup>27</sup><https://docs.docker.com/desktop/windows/install/>

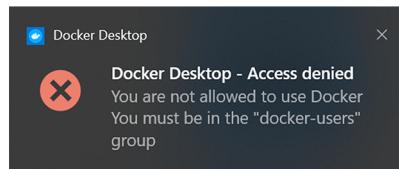
<sup>28</sup><https://docs.microsoft.com/en-us/windows/wsl/install-win10>

<sup>29</sup><https://docs.microsoft.com/windows/wsl/wsl2-kernel>

<sup>30</sup><https://hub.docker.com/editions/community/docker-ce-desktop-windows/>

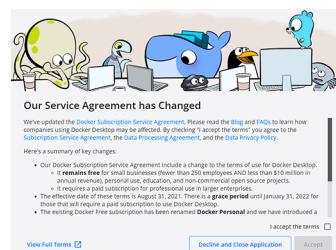
Al iniciar el proceso de instalación Docker evalua si tiene configurado correctamente WSL 2 y le pide instalar los componentes de Windows necesarios para utilizarlo. Garantice que la casilla de selección esté marcada, luego dele al botón de aceptar. El proceso debe demorar menos de un minuto y al final le muestra el mensaje “Instalation succeeded”.

Localice e inicie la aplicación Docker Desktop desde el menú de inicio. Es posible que aparezca un mensaje diciendo que no tiene permisos para utilizar Docker. Si le sucede esto debe ir a la “*Administración de equipos > Usuarios y grupos locales > Usuarios*”, seleccione su usuario y agregue el grupo *docker-users* en la sección “*Miembro de*”. Reinicie el ordenador y vuelva a abrir la aplicación *Docker Desktop*.



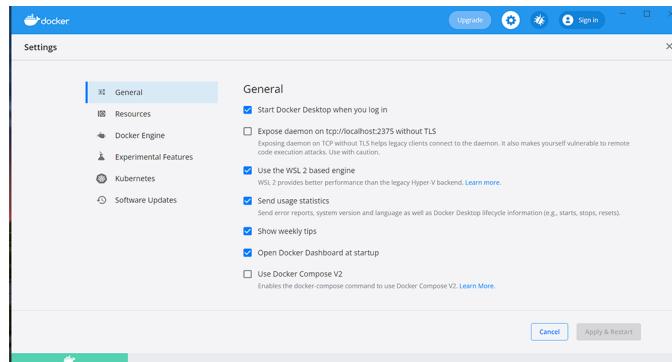
Error de permisos al iniciar Docker

Una vez iniciado el sistema aparece la ventana de términos y condiciones definidos por Docker. Esta ventana está relacionada con los aspectos vistos en la sección [Precios y suscripciones](#). En su caso puede aceptar sin problemas porque en su condición de estudiante no tiene que realizar ningún pago.



Términos y condiciones de Docker

Al aceptar los términos y condiciones Docker inicia sus procesos, abre la ventana principal con el Dashboard y le brinda la opción de conocer algunos *tips* y sugerencias. Si lo desea, tómese unos minutos conociendo las últimas características incluidas en la aplicación, pero luego pase a revisar si está activado WSL 2 en la sección *Settings*.



### WSL2 habilitado en Docker Desktop

Ahora que está confirmado el uso de WSL 2 puede iniciararlo y ejecutar el primer comando de Docker para conocer la versión que ha sido instalada.

```
$ docker version

Client: Docker Engine - Community
Cloud integration: v1.0.25
Version:           20.10.16
API version:      1.41
Go version:       go1.17.10
Git commit:       aa7e414
Built:            Thu May 12 09:17:39 2022
OS/Arch:          linux/amd64
Context:          default
Experimental:    true

Server: Docker Desktop
Engine:
  Version:           20.10.16
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.17.10
  Git commit:       f756502
  Built:            Thu May 12 09:15:42 2022
  OS/Arch:          linux/amd64
  Experimental:    false
containerd:
  Version:           1.6.4
  GitCommit:        212e8b6fa2f44b9c21b2798135fc6fb7c53efc16
runc:
  Version:           1.1.1
```

```
GitCommit:      v1.1.1-0-g52de29d
docker-init:
Version:        0.19.0
GitCommit:      de40ad0
```

No pase por alto el siguiente detalle, Docker le brinda la versión de ambos componentes, el cliente y el servidor, ambos elementos fueron explicados en la sección [Arquitectura y Componentes](#).

Si no desea conocer los procesos de instalación para otras distribuciones puede saltar directamente a la sección [Mi primer contenedor](#).

## Docker Desktop para Linux

Docker Desktop para Linux (DD4L) es la aplicación de interfaz gráfica diseñada para instalar y utilizar Docker en distribuciones Linux. Las plataformas habilitadas hasta el momento son Ubuntu, Fedora y Debian, pero siempre puede [consultar el sitio oficial<sup>31</sup>](#) por si existen nuevas opciones. Si su sistema operativo no corresponde con las opciones disponibles debe pasar a la siguiente sección [Docker Engine para Linux](#).

## Requisitos previos

Antes de instalar DD4L compruebe que cumple con los siguientes requerimientos:

- \* Kernel de 64 bit y habilitado para virtualización con KVM.
- \* Tener instalado QEMU 5.2 o una versión superior.
- \* Entorno gráfico Gnome o KDE.
- \* Tener mínimo 4GB de RAM.

DD4L funciona en una máquina virtual dentro de su ordenador, de aquí la necesidad de tener habilitado la virtualización. Si desea conocer los motivos de este funcionamiento consulte el [enlace en el sitio oficial<sup>32</sup>](#).

Confirme que tiene habilitado correctamente los módulos de KVM antes de iniciar la instalación.

```
$ lsmod | grep kvm
```

kvm_amd	167936	0
ccp	126976	1 kvm_amd
kvm	1089536	1 kvm_amd
irqbypass	16384	1 kvm

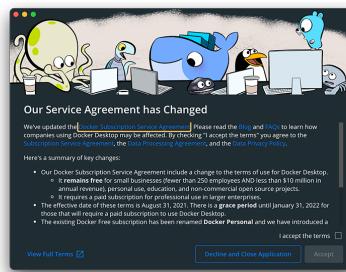
<sup>31</sup><https://docs.docker.com/desktop/linux/install/#supported-platforms>

<sup>32</sup><https://docs.docker.com/desktop/linux/install/#why-docker-desktop-for-linux-runs-a-vm>

## Instalación de DD4L

Descargue el fichero de instalación correspondiente con su distribución desde el [sitio oficial<sup>33</sup>](#). Luego siga los pasos de la instalación según corresponda, [enlace<sup>34</sup>](#). Es muy posible que necesite instalar alguna dependencia adicional durante el proceso, si es el caso identifique cuáles son a través de los mensajes de su Terminal y complete la guía.

Después de instalar Docker Desktop solamente queda buscar la aplicación instalada e iniciar su ejecución. La primera pantalla en mostrarse será la de términos y condiciones, la cual hace referencia a las suscripciones analizadas en la secciones anteriores.



Docker términos y condiciones

Una vez aceptados los términos y condiciones aparece el *Dashboard* o cuadro de mando de Docker. Utilice unos minutos para explorar las opciones existentes o ajustar los recursos utilizados, luego pasemos a ejecutar el primer comando con `docker` para conocer la versión que ha sido instalada.

Abra su Terminal y ejecute el siguiente comando.

```
$ docker version
```

```
Client: Docker Engine - Community
Version:           20.10.14
API version:      1.41
Go version:       go1.16.15
Git commit:       a224086
Built:            Thu Mar 24 01:47:58 2022
OS/Arch:          linux/amd64
Context:          desktop-linux
Experimental:    true
```

```
Server: Docker Desktop 4.9.1 (81317)
```

```
Engine:
Version:          20.10.16
```

<sup>33</sup><https://docs.docker.com/desktop/linux/install/>

<sup>34</sup><https://docs.docker.com/desktop/linux/install/#generic-installation-steps>

```
API version:      1.41 (minimum version 1.12)
Go version:       go1.17.10
Git commit:       f756502
Built:            Thu May 12 09:15:42 2022
OS/Arch:          linux/amd64
Experimental:    false
containerd:
  Version:        1.6.4
  GitCommit:      212e8b6fa2f44b9c21b2798135fc6fb7c53efc16
runc:
  Version:        1.1.1
  GitCommit:      v1.1.1-0-g52de29d
docker-init:
  Version:        0.19.0
  GitCommit:      de40ad0
```

No pase por alto el siguiente detalle, Docker le brinda la versión de ambos componentes, el cliente y el servidor, ambos elementos fueron explicados en la sección [Arquitectura y Componentes](#).

Si no desea conocer los procesos de instalación para otras distribuciones puede saltar directamente a la sección [Mi primer contenedor](#).

## Docker Engine para Linux

[Docker Engine](#)<sup>35</sup> agrupa tanto el cliente (`docker`) como el servidor (`dockerd`). Son las dependencias mínimas necesarias para utilizar contenedores en un sistema operativo. *Docker Engine* es todo lo que necesita cuando no desee, o no pueda, utilizar Docker Desktop. Docker Engine es la forma recomendada de instalar Docker en servidores de producción o entornos sin interfaz gráfica.

Docker Engine puede ser instalado en una gran variedad de distribuciones Linux, donde cada una tiene diferencias entre sí dependiendo de los comandos y tipos de paquetes que utilizados. Si se dedica un espacio en el libro para cada distribución puede resultar aburrido, o peor aún, pudiese faltar la distribución que usted necesite. Por tal motivo, se va a utilizar un [script creado por Docker](#)<sup>36</sup> con el que puede realizar la instalación en la mayoría de distribuciones.

En algún caso aislado pudiese fallar el script, si es este su caso le recomiendo revisar los siguientes puntos:

- Consulte los pasos necesarios para la instalación en la distribución que esté utilizando: [CentOS](#)<sup>37</sup>,

---

<sup>35</sup><https://docs.docker.com/engine/>

<sup>36</sup><https://github.com/docker/docker-install>

<sup>37</sup><https://docs.docker.com/engine/install/centos/>

Debian<sup>38</sup>, Fedora<sup>39</sup>, RHEL<sup>40</sup> o Ubuntu<sup>41</sup>

- En caso de utilizar sistemas derivados de Debian (*BunsenLabs Linux*, *Kali Linux* or *LMDE*) o Ubuntu (*Kubuntu*, *Lubuntu* o *Xubuntu*) debe hacer uso las guías de las distribuciones de origen.
- Consulte la lista general de distribuciones<sup>42</sup> permitidas por Docker para saber si la suya está incluida.
- Si ninguna de las anteriores opciones ha funcionado póngase en contacto con el autor del libro<sup>43</sup>.

## Instalación y puesta en marcha

Docker brinda el script [get.docker.com](https://get.docker.com)<sup>44</sup> para realizar instalaciones en entornos de desarrollo de forma rápida, sin embargo *no se recomienda utilizar este script para entornos de producción*. Siempre será más seguro seguir paso a paso las indicaciones específicas de cada distribución por si desea adaptar la instalación a algún requerimiento que tenga en sus servidores.

Como recomendación general siempre debe examinar los scripts descargados antes de instalarlos. En este caso el script de Docker tiene las siguientes características:

- Necesita de privilegios `sudo` para su ejecución.
- Detecta la distribución de Linux y configura el gestor de paquetes para la instalación.
- Instala las dependencias necesarias para el funcionamiento de Docker sin pedir confirmación.
- Instala la última versión estable de `docker`, `containerd` y `runc`.
- El script no está diseñado para hacer actualizaciones.

Le recomiendo ejecutar el script en modo *DRY* para conocer las configuraciones a realizar en su ordenador.

```
$ curl -fsSL https://get.docker.com -o get-docker.sh  
$ DRY_RUN=1 sh ./get-docker.sh
```

Examine los comandos para detectar comportamientos no deseados. Si está de acuerdo con el procedimiento repita el comando sin la variable `DRY_RUN`:

```
$ sudo sh ./get-docker.sh
```

Al finalizar la instalación el servicio de `docker` es iniciado automáticamente y se comprueba la versión instalada en el sistema. En las distribuciones con base RPM como CentOS, Fedora, RHEL o SLES tiene que iniciarlos manualmente con los comandos `systemctl` o `service` según corresponda.

<sup>38</sup><https://docs.docker.com/engine/install/debian/>

<sup>39</sup><https://docs.docker.com/engine/install/fedora/>

<sup>40</sup><https://docs.docker.com/engine/install/rhel/>

<sup>41</sup><https://docs.docker.com/engine/install/ubuntu/>

<sup>42</sup><https://docs.docker.com/engine/install/>

<sup>43</sup>[https://leanpub.com/erase-una-vez-docker/email\\_author/new](https://leanpub.com/erase-una-vez-docker/email_author/new)

<sup>44</sup><https://get.docker.com/>

```
$ sudo systemctl start docker
```

(Opcional) Utilice los siguientes comandos para habilitar el inicio automático de Docker en aquellos sistemas operativos que lo necesiten.

```
$ sudo systemctl enable docker.service  
$ sudo systemctl enable containerd.service
```

Una vez terminada la instalación, consulte la versión existente en la máquina.

```
$ docker version
```

```
Client: Docker Engine - Community  
Version:           20.10.10  
API version:      1.41  
Go version:       go1.16.9  
Git commit:       b485636  
Built:            Mon Oct 25 07:42:59 2021  
OS/Arch:          linux/amd64  
Context:          default  
Experimental:    true
```

```
Server: Docker Engine - Community  
Engine:  
  Version:           20.10.10  
  API version:      1.41 (minimum version 1.12)  
  Go version:       go1.16.9  
  Git commit:       e2f740d  
  Built:            Mon Oct 25 07:41:08 2021  
  OS/Arch:          linux/amd64  
  Experimental:    false  
containerd:  
  Version:           1.4.11  
  GitCommit:        5b46e404f6b9f661a205e28d59c982d3634148f8  
runc:  
  Version:           1.0.2  
  GitCommit:        v1.0.2-0-g52b36a2  
docker-init:  
  Version:           0.19.0  
  GitCommit:        de40ad0
```

No pase por alto el detalle donde Docker le brinda la versión de ambos componentes, el cliente y el servidor. Recuerde que ambos elementos fueron explicados en la sección [Arquitectura y Componentes](#).

## Acciones después de instalar Docker

Hasta el momento ha sido necesario utilizar el usuario *root* o el comando *sudo* para instalar y ejecutar Docker, pero también va ser necesario ejecutar comandos con usuarios *no-root*.

Las buenas prácticas de seguridad explican que todo sistema o servicio debe tener los mínimos permisos posibles para su funcionamiento, ni más ni menos. Partiendo de esta premisa, el usuario que utilice en los servidores de producción va a ser *no-root*. Sin embargo, este usuario va a necesitar algo más de configuración para poder utilizar los comandos de Docker.

Intente conocer la versión de Docker con un usuario no-root. De inmediato va a notar el mensaje de error por la falta de permisos para acceder al *socket* del servidor *dockerd*.

```
$ docker version

Client: Docker Engine - Community
Version:           20.10.10
API version:      1.41
Go version:       go1.16.9
Git commit:       b485636
Built:            Mon Oct 25 07:42:59 2021
OS/Arch:          linux/amd64
Context:          default
Experimental:    true

Got permission denied while trying to connect to the Docker daemon socket at unix:///var/\run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/version": dial unix /var/ru\n/docker.sock: connect: permission denied
```

Para utilizar usuarios *no-root* debe crear un grupo *Unix* con el nombre *docker* y luego adicionar su usuario a este grupo. Utilice los siguientes comandos para realizar esta configuración.

Crear grupo Unix. Puede darse el caso donde el grupo haya sido creado durante el proceso de instalación. En ese caso puede seguir con el próximo comando.

```
$ sudo groupadd docker
```

Añadir su usuario al grupo *docker*.

```
$ sudo usermod -aG docker $USER
```

Ahora debe salir y entrar del Terminal con su usuario para hacer efectivo el cambio. Después de esta acción vuelva a utilizar el comando `docker version` y compruebe que el error ha desaparecido.

## Mi primer contenedor

En casi todas las herramientas existe un ejemplo sencillo de hacer para mostrar el correcto funcionamiento del sistema, el conocido *Hola Mundo!*. Como no podía ser menos, en esta sección tiene lugar el ejemplo *Hola Mundo!* para Docker, y el comando a utilizar es el siguiente:

```
$ docker container run hello-world
```

En este simple comando se pueden ver muchos de los conceptos asociados a los contenedores, disfrute de este primer momento donde pone en funcionamiento su primer contenedor.

El mensaje mostrado en pantalla va a ser similar a:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
93288797bd35: Pull complete
Digest: sha256:975f4b14f326b05db86e16de00144f9c12257553bba9484fed41f9b6f2257800
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
(arm64v8)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

```
https://hub.docker.com/
```

For more examples and ideas, visit:

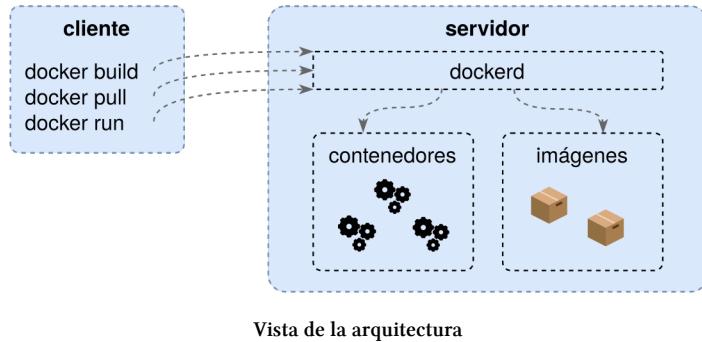
```
https://docs.docker.com/get-started/
```

Si ha recibido este mensaje significa que todo está funcionando a las mil maravillas. ¡Enhорabuena por su primer contenedor!

En los próximos capítulos se van a explicar los detalles de lo sucedido para lograr obtener este resultado.

## Resumen del capítulo

Durante el capítulo se ha explicado la arquitectura de Docker, modelo cliente - servidor, utilizada para gestionar las imágenes y los contenedores.



Se explicaron las suscripciones existentes en Docker, así como también cuál debe ser utilizada según el entorno. En este punto, lo más importante es recordar que al estar como estudiante no tiene que realizar ningún pago por el uso de Docker. Puede seguir utilizándolo con toda confianza.

Se mostraron los pasos a seguir a la hora de instalar Docker en Windows, Mac y Linux. Cada sistema operativo tiene sus características, por lo tanto siempre existen diferencias entre ellos, pero lo que es común es el poder utilizar Docker Desktop como herramienta para el desarrollo de aplicaciones y microservicios utilizando en contenedores. Por otro lado, para entornos sin interfaz gráfica se recomienda instalar Docker Engine.

Después de la instalación de Docker se puso en funcionamiento el primer contenedor “*Hola Mundo*”.

Este viaje no ha hecho más que empezar, con Docker instalado en su ordenador no hay nada que le impida avanzar con los contenedores. Tenga listo su Terminal y siga al próximo capítulo.

# Primeros pasos con Docker

La instalación de Docker es la llave a un nuevo mundo de comandos, de repente tiene cientos de opciones para gestionar contenedores. Es normal tener dudas por donde empezar, pero pasa eso estamos aquí, para dar los primeros pasos poco a poco.

El punto de partida va ser comprender la estructura de comandos propuesta por Docker. Es imposible aprenderse todos los comandos de memoria, pero si puede entender cómo están agrupados y a cuál concepto dan solución.

Luego se pasa a explicar en detalle el ejemplo “*Hola Mundo*”. La explicación permite llevar a la práctica los conceptos de imágenes y contenedores a través de las rutinas de Docker.

Conocerá también los registros de imágenes, cuáles son los más utilizados y cómo realizar la búsqueda de imágenes. Son muchos los registros de imágenes que existen actualmente, pero en este capítulo serán mencionados los más relevantes.

De igual forma aprenderá a interpretar la estructura del nombre de las imágenes. Con este conocimiento podrá saber de forma rápida dónde está almacenada la imagen, el usuario que la ha creado y la versión del software que lleva dentro. Después de conocer la estructura del nombre de las imágenes pasará a dominar los comandos básicos para gestionar imágenes y contenedores. Es este punto se hace referencia a acciones muy repetidas como son listar, eliminar y descargar contenidos.

Por último, pero no menos importante, aprenderá que existen múltiples variantes para manipular la información de salida de los comandos de Docker. A través del parámetro `--format` podrá dar rienda suelta a su imaginación en el trabajo con los contenedores.

Sin más dilación, ¡pasemos al primer tema!

## Estructura de comandos

Docker cuenta con cientos de comandos en el Terminal si se tienen en cuenta todas las posibles combinaciones (*imágenes, contenedores, redes, volumes, etc*). En un inicio esto puede parecer abrumador, pero el objetivo de esta sección es lograr identificar el mecanismo que permita saber cómo llegar al comando deseado sin tener que memorizarlos todos.

Para lograr este objetivo lo primero va a ser dominar la ayuda que brinda Docker a través del parámetro `--help`. Con este parámetro puede conocer los principales subcomandos existentes. Va a ser importante tener esta opción en la mente todo el tiempo, porque Docker se actualiza múltiples veces en el año, y seguir al detalle todos sus cambios puede ser difícil. De ahí la necesidad de consultar la ayuda siempre que necesite saber las opciones disponibles.

```
$ docker --help | less
```

El comando `| less` le permite ver todos los comandos de Docker desde el inicio, pudiendo desplazarse hacia arriba y hacia abajo según necesite. Veamos a continuación cómo interpretar el resultado de utilizar este primer comando.

## Opciones generales para todos los comandos

Seguido al comando `docker` puede utilizar los parámetros que se muestran en la sección *Options*. Estos elementos no están asociados a un concepto específico, sin embargo modifican el comportamiento de la herramienta, como por ejemplo, habilitar el modo *Debug*, indicar la ubicación de la configuración de `docker` o utilizar un servidor externo, por solo citar algunas opciones.

```
Usage: docker [OPTIONS] COMMAND
A self-sufficient runtime for containers

Options:
  --config string          Location of client config files ...
  -c, --context string     Name of the context to use to ...
  -D, --debug               Enable debug mode
  -H, --host list           Daemon socket(s) to connect to
  -v, --version              Print version information and quit
...
...
```

No se desespere si no sabe cómo llevarlos a la práctica en este momento, porque durante el transcurso de los capítulos se van a utilizar de una u otra forma.

## Comandos de gestión

Los comandos de gestión agrupan los conceptos existentes en Docker. Detrás de cada comando de gestión existen nuevos comandos especializados. Entre los grupos más utilizados están *image* y *container*.

```
Usage: docker [OPTIONS] COMMAND
...
Management Commands:
builder      Manage builds
buildx*      Docker Buildx (Docker Inc., v0.7.1)
compose*     Docker Compose (Docker Inc., v2.2.3)
config       Manage Docker configs
container    Manage containers
context      Manage contexts
image        Manage images
...
```

Para conocer los comandos que se encuentran detrás de un grupo de gestión puede utilizar el parámetro `--help`, pero esta vez después del comando de gestión. El ejemplo que se muestra a continuación corresponde al comando de gestión *image*.

```
$ docker image --help

Usage: docker image COMMAND

Manage images

Commands:
build      Build an image from a Dockerfile
history   Show the history of an image
import    Import the contents from a tarball to create a filesystem image
inspect   Display detailed information on one or more images
load      Load an image from a tar archive or STDIN
ls        List images
...
```

Como puede observar, las acciones mostradas corresponden con el concepto de imagen, como por ejemplo: construir, obtener, listar o importar.

## Comandos individuales

Después de los comandos de gestión viene la sección *Commands*. En este apartado va a encontrar comandos que fueron incluidos en las primeras versiones de Docker, se puede imaginar que estamos hablando de hace más de ocho años.

En las primeras versiones de Docker existían pocos comandos y estaban todos el mismo nivel, pero al pasar el tiempo nuevas funcionalidades llegaron y fue necesario encontrar la forma de organizarlos, de ahí que aparecieron los *Comandos de gestión*. En aras de mantener la compatibilidad con los sistemas

que estaban en funcionamiento Docker tomó la decisión de mantener los comandos iniciales, pero los nuevos comandos fueron organizadas en grupos según su función.

Como resultado final puede ver funciones en Docker que tienen asociadas dos comandos con el mismo resultado. Por ejemplo, si desea listar los contenedores puede:

Comando directo:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

O utilizar la función del comando de gestión:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Entonces, *¿cuál de los dos comandos se debe utilizar?*

Mi recomendación es utilizar los comandos de gestión siempre que sea posible. Al estar asociados a conceptos de Docker es más fácil identificar la función que realiza. Otro elemento a tener en cuenta es que todas las funciones de los comandos independientes están incluidas en los comandos de gestión, pero *no* es así a la inversa. Por lo tanto, siempre va a tener que utilizar los comandos de gestión de una u otra forma

## Documentación oficial de Docker

Además del parámetro `--help` está la opción de consultar la ayuda en el sitio oficial de Docker. En ella se encuentran las descripciones y el propósito de cada comando, así como interesantes ejemplos que pueden ser de mucha ayuda.

<https://docs.docker.com/engine/reference/commandline/docker/>

Dentro del sitio web utilice el menú de la izquierda para navegar por los múltiples comandos que tiene Docker. Mi sugerencia es dedicar unos minutos a revisar el sitio para familiarizarse con su contenido.

## Hola Mundo en detalles

En el capítulo anterior se realizó el ejemplo *Hola Mundo* para contenedores. El objetivo fue comprobar la correcta instalación de `docker`, pero quedó pendiente explicar el significado de aquellas primeras líneas impresas en pantalla.

Como lo prometido es deuda, ha llegado el momento de explicar lo sucedido en aquel comando. No se preocupe si no recuerda el ejemplo, vuelva a mirarlo a continuación:

```
$ docker container run hello-world  
  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
93288797bd35: Pull complete  
Digest: sha256:975f4b14f326b05db86e16de00144f9c12257553bba9484fed41f9b6f2257800  
Status: Downloaded newer image for hello-world:latest  
  
Hello from Docker!  
...
```

Siguiendo la estructura de comandos puede imaginar lo sucedido. Se le indica a *docker* que inicie un contenedor (`container run`) utilizando la imagen `hello-world`. *Docker* busca la imagen en su ordenador, pero al no encontrarla sale a Internet para descargarla.

```
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world
```

Al terminar la descarga *docker* muestra el código sha256 de la imagen, este código es único para cada una.

```
93288797bd35: Pull complete  
Digest: sha256:975f4b14f326b05db86e16de00144f9c12257553bba9484fed41f9b6f2257800  
Status: Downloaded newer image for hello-world:latest
```

Después de descargar la imagen, *docker* inicia un contenedor a partir de ella. El contenedor iniciado imprime en pantalla el texto del saludo, cuando termina de imprimirlo el contenedor se detiene automáticamente.

```
Hello from Docker!  
This message shows that your installation appears to be working correctly.  
...
```

El siguiente listado resume los pasos realizados por *docker*.

1. Se busca la imagen en la máquina donde se ha ejecutado el comando.
2. (Opcional) Se descarga la imagen desde Internet si no es encontrada.
3. Se inicia un contenedor a partir de la imagen.
4. El contenedor realiza las acciones configuradas.
5. El contenedor se detiene.

A grandes rasgos es un procedimiento sencillo y fácil de entender. Sin embargo, hay mucho trabajo por detrás para garantizar que las cosas difíciles parezcan fáciles. Dominar este flujo es de vital importancia para seguir avanzando en el mundo de los contenedores.

## Registros de imágenes

De la sección anterior quedó un misterio sin resolver, *¿de dónde se ha descargado Docker la imagen hello-world?*

Todas las imágenes de contenedores se guardan en almacenes digitales llamados *Registros*. Uno de los registros más utilizados es [Docker Hub](#)<sup>45</sup> y ha sido desde este lugar donde Docker ha descargado la imagen `hello-world`.

En *Docker Hub* existen cientos de miles de imágenes. Piense en una herramienta o sistema que utilice a diario y búsqueda en *Docker Hub*, va a tener una gran probabilidad de que exista. La mayoría de comunidades comparten sus productos como contenedores en esta gran plataforma.

Las imágenes en *Docker Hub* pueden ser buscadas de dos formas, a través de su sitio web o desde el Terminal con líneas de comandos. Veamos a continuación ambos procedimientos.

### Buscar imágenes desde el sitio web Docker Hub

Utilice el siguiente enlace para acceder a *Docker Hub*:

<https://hub.docker.com/search?type=image>

Arriba del todo va a encontrar el panel de búsquedas, escriba el nombre de la tecnología y observe los resultados. También puede hacer uso de los filtros en la columna de la izquierda para reducir los resultados a una temática específica. A modo de prueba busque el término `hello-world`, acto seguido debe aparecer un listado con muchas imágenes que tienen la frase `hello-world` en el nombre, pero no se preocupe, la que estamos buscando tiene la etiqueta **Official Image**.

The screenshot shows the Docker Hub search interface. At the top, there's a blue header bar with the Docker logo, navigation links for 'Explore', 'Pricing', 'Sign In', and a 'Register' button. Below the header, a search bar contains the query 'hello-world'. To the right of the search bar is a dropdown menu set to 'Best Match'. The main content area displays a list of search results. The first result is for the official 'hello-world' image, which has been updated 11 days ago. It shows the repository name 'hello-world', the Docker logo, and the text 'DOCKER OFFICIAL IMAGE'. To the right of the repository details are metrics: '1B+' downloads and '1.8K' stars. Below the repository info is a brief description: 'Hello World! (an example of minimal Dockerization)'. Underneath the description is a horizontal row of supported host operating systems: Windows, Linux, IBM Z, ARM, ARM 64, 386, mips64le, PowerPC 64 LE, x86-64, and riscv64.

Sitio web Docker Hub

Una vez identificada la imagen acceda a ella para ver los detalles de su funcionamiento, su última fecha de actualización y los comentarios de los usuarios que la han utilizado. Dedique unos minutos

<sup>45</sup><https://hub.docker.com/>

en descubrir los elementos presentes en ella, note que no es necesario tener una cuenta en la plataforma Docker Hub para realizar las búsquedas.

## **Etiquetas definidas en Docker Hub**

El equipo de Docker ha ido creando etiquetas para clasificar las imágenes almacenadas en el Hub. Esta clasificación permite a los usuarios saber de forma rápida algunas características de la imagen buscada. En este momento existen tres etiquetas:

- **Official Image<sup>46</sup>**: la imagen está bajo el mantenimiento de la empresa Docker. Esta etiqueta brinda garantía de calidad, buenas prácticas y actualización.
- **Verified Publisher<sup>47</sup>**: la imagen es publicada y mantenida por la entidad comercial dueña del producto. La imagen cuenta con garantía de seguridad y mantenimiento por parte del dueño.
- **Open source program<sup>48</sup>**: identifica los proyectos alineados con la comunidad open source con fines no comerciales. Docker les brinda apoyo a estos proyectos en la plataforma para que puedan seguir creciendo y desarrollándose.

Las imágenes sin etiquetas también pueden ser utilizadas, no hay ningún problema en esto. La diferencia radica que debe ser usted quien analice la calidad y mantenimiento del producto.

## **Buscar imágenes desde el Terminal**

La búsqueda de imágenes también puede ser realizada a través del terminal y el comando reservado para esta función es `search`. Realice la búsqueda de la imagen `hello-world` para ver los resultados del comando.

```
$ docker search hello-world
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
hello-world	Hello World! (an...)	1630	[OK]	
kitematic/hello-world-nginx	A light-weight n...	151		
tutum/hello-world	Image to test do...	87	[OK]	
dockercloud/hello-world	Hello World!	19	[OK]	
crccheck/hello-world	Hello World web	15	[OK]	
vadimo/hello-world-rest	A simple REST Se...	5	[OK]	
ppc64le/hello-world	Hello World! (an...	2		

El comando `search` realiza la búsqueda solamente en el registro *Docker Hub*. Puede identificar en los resultados algunos de los términos ya explicados en los párrafos anteriores, como por ejemplo *Official*.

---

<sup>46</sup>[https://docs.docker.com/docker-hub/official\\_images/](https://docs.docker.com/docker-hub/official_images/)

<sup>47</sup><https://docs.docker.com/docker-hub/publish/>

<sup>48</sup><https://www.docker.com/community/open-source/application/>

Al igual que con el resto de comandos, puede hacer uso del parámetro `--help` o de su [referencia en el sitio web<sup>49</sup>](#) para ampliar la información sobre `search`.

Un ejemplo interesante puede ser filtrar la búsqueda y mostrar solamente las imágenes oficiales. Utilice el comando que aparece a continuación para lograrlo.

```
$ docker search --filter=is-official=true hello-world
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
hello-world	Hello World! (an ...	1630	[OK]	

## ¿Cuántos registros de imágenes existen?

A día de hoy existen muchos registros de imágenes. *Docker Hub* fue el primero de todos, pero poco a poco las plataformas han ido creando servicios similares, p.ej, [GitHub<sup>50</sup>](#), [GitLab<sup>51</sup>](#), [AWS<sup>52</sup>](#), [GCP<sup>53</sup>](#), [Azure<sup>54</sup>](#) y [Digital Ocean<sup>55</sup>](#), por solo nombrar algunos.

Usted como usuario puede decidir la plataforma que desee, pero antes de hacerlo, consulte las condiciones y características de cada una para conocer las opciones que ofrecen. Los siguientes tres elementos forman parte de las posibles opciones a encontrar:

- Precio por almacenar las imágenes.
- Número de imágenes públicas y privadas (repositorios).
- Velocidad de transferencia para la entrada y salida.

Debe valorar todos los elementos de conjunto y tomar una decisión al respecto. Esta no es una decisión crítica, en el momento que lo deseé puede cambiar a otro registro si lo considera necesario.

## Los nombres de las imágenes

Los nombres de las imágenes reflejan su origen. Cada uno muestra la información donde está almacenada, a quién pertenece y cuál versión se está utilizando. La estructura de nombre sigue el siguiente patrón.

---

<sup>49</sup><https://docs.docker.com/engine/reference/commandline/search/>

<sup>50</sup><https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry>

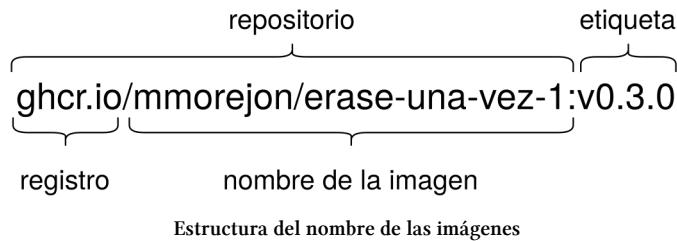
<sup>51</sup>[https://docs.gitlab.com/ee/user/packages/container\\_registry/](https://docs.gitlab.com/ee/user/packages/container_registry/)

<sup>52</sup><https://aws.amazon.com/es/ecr/>

<sup>53</sup><https://cloud.google.com/container-registry>

<sup>54</sup><https://azure.microsoft.com/es-es/services/container-registry/>

<sup>55</sup><https://www.digitalocean.com/products/container-registry>



A continuación aparece una breve descripción para cada elemento.

## Registro

El registro es donde está almacenada la imagen. La dirección *web* utilizada brinda una pista del lugar o ubicación del registro, p.ej:

URL	Ubicación / Plataforma
index.docker.io	Docker Hub Registry
ghcr.io	GitHub Container Registry
<account-id>.dkr.ecr.us-east-1.amazonaws.com	AWS Container Registry
registry.digitalocean.com	DigitalOcean Container Registry

Existen muchos otros registros, pero usted podrá identificarlos poco a poco según lo necesite.

## Nombre

El nombre de la imagen permite identificar de forma única el elemento dentro del registro. Muchas de las plataformas públicas incluyen el identificador del usuario como prefijo al nombre de la imagen, de esta manera permiten a diferentes usuarios tener el mismo nombre de imagen sin dejar de ser únicas en la plataforma.

En la imagen anterior se puede observar el uso de `mmorejon` delante del nombre `erase-una-vez-1`, teniendo como resultado de nombre `mmorejon/erase-una-vez-1`.

## Repositorio

El repositorio es la unión del registro y el nombre. Este valor es el mostrado en la columna *REPOSITORY* cada vez que se listan las imágenes con el comando *docker*.

Aunque esta sea la regla general, existen casos interesantes que no siguen este formato. El primero corresponde con las imágenes oficiales en Docker Hub. Estas imágenes utilizan la palabra `library` en vez del identificador del usuario, y no muestran el registro ni el prefijo en la columna *REPOSITORY*. p.ej.

```
$ docker image ls nginx
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	1.21.6	9c1ff20ac9c9	2 months ago	134MB
nginx	1.20.2	e08034998ac8	3 months ago	134MB

## Etiqueta

Las imágenes de Docker, al igual que los sistemas de software, pueden tener múltiples versiones, y cada versión es marcada con una etiqueta (*tag*).

La intención de etiquetar las imágenes corresponde con el deseo de asociar cada imagen con la versión del producto que lleva dentro. Recuerde que el propósito principal de Docker, y en general de los contenedores, es empaquetar, distribuir y ejecutar aplicaciones, por lo tanto es muy conveniente hacer uso de las etiquetas para saber cuál versión de la aplicación se está ejecutando.

Ejemplos de etiquetas: v0.3.0, latest, alpine.

En el mundo de los contenedores la etiqueta `latest` está reservada para representar la ausencia de etiquetas. En el momento que se incluya una imagen en el repositorio, si no se especifica una etiqueta, Docker incluirá automáticamente `latest` como parte del nombre. De esta forma se garantiza la existencia de una etiqueta en la estructura del identificador.

A la hora de descargar las imágenes se utiliza la misma filosofía. Si no se incluye una etiqueta se utiliza `latest` por defecto.

## Gestión de imágenes y contenedores

Saber acceder a Docker Hub le permite buscar las imágenes de su interés para experimentar con ellas luego. Sin embargo, primero debe dominar los comandos de gestión para imágenes y contenedores. Saquemos provecho del ejemplo *Hola Mundo* para conocer algunos de los comandos más utilizados en ambos conceptos.

### Comandos asociado a las imágenes

El primer comando va a ser listar las imágenes existentes en la máquina. En el resultado va a poder ver datos interesantes asociados a cada imagen, p.ej: el tamaño que ocupa en disco y la fecha de creación. Si tuviese muchas imágenes descargadas puede verlas todas con este comando.

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	18e5af790473	4 months ago	9.14kB

Al ver el resultado le puede resultar familiar la columna *REPOSITORY* donde se muestra el identificador de la imagen (*registro + nombre*), así como también la columna *TAG*, haciendo referencia a la versión de a la imagen.

El próximo comando a mostrar se utiliza para descargar imágenes. Es necesario conocer el identificador de la imagen antes de utilizarlo. En el ejemplo que se muestra a continuación se descargan tres imágenes, dos versiones de Nginx en Docker Hub y la aplicación *erase-una-vez-1* ubicada en el registro de GitHub.

```
$ docker image pull ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
$ docker image pull nginx:1.20.2
$ docker image pull nginx:1.21.6
```

Liste nuevamente las imágenes cuando termine la descarga y compruebe que están todas las imágenes mencionadas en el comando anterior. Docker también brinda la opción de filtrar el listado utilizando el valor de la columna *REPOSITORY*. Si conoce el nombre del repositorio que está buscando agréguelo después del comando *list*.

```
$ docker image list nginx
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	1.20.2	e08034998ac8	11 days ago	134MB
nginx	1.21.6	2e7e2ec411a6	11 days ago	134MB

En este momento ya sabe buscar, descargar y listar las imágenes, pero para cerrar el ciclo de gestión de imágenes solamente falta saber eliminarlas. Siguiendo la filosofía de comandos, no le va ser difícil entender la siguiente instrucción para borrar imágenes:

```
$ docker image rm nginx:1.20.2
```

```
Untagged: nginx:1.20.2
Untagged: nginx@sha256:02923d65cde08a49380ab3f3dd2f8f90aa51fa2bd358bd85f89345848f6e6623
Deleted: sha256:e08034998ac81e1cf75ce5bd3fc94630787e21d246cd5717792792d1bc84f552
Deleted: sha256:1c56e3672ee621273351c6a99d25cddae716b92d37d34d82794bde5f9ced57e2
Deleted: sha256:e4ea413e433348a39aae1661bd2088d381809247b2c0f04f45b6d0b2d8e55290
Deleted: sha256:9ea9d1dae7621e94766874a9cebad149a8ba331f40c427a46fd3e132cf0698b7
Deleted: sha256:8d7af4642159890ee64f7b6508f5cb82f9e3f8a6598f529d8103498430af5a6d
Deleted: sha256:057b3b84edcb5fe9c5d2b2828e306dbb734423a3a49d6e4c8bb15e2f64101d10
```

De esta manera puede eliminar las imágenes y liberar el espacio en disco. El único requisito es conocer el nombre completo de la imagen que deseé borrar. Liste nuevamente las imágenes para comprobar que `nginx:1.20.2` ha sido eliminada de la máquina.

Existen muchos otros comandos interesantes relacionados con las imágenes, pero recuerde que el capítulo se llama *Primeros pasos con Docker*, por lo tanto este es solo el comienzo. A medida que avancen los capítulos se van a ir mostrados nuevos comandos y funciones de Docker.

## Comandos asociados a los contenedores

Al igual que las imágenes, los contenedores tienen asociados múltiples comandos que permiten su gestión. Hay algunos más utilizados que otros, pero sin dudas todos son necesarios.

Listar elementos es una de las acciones más repetidas. El comando `ls` permite listar los contenedores, pero por defecto lista solamente aquellos que estén funcionando (*status running*).

```
$ docker container ls
```

En el ejemplo realizado con la imagen `hello-world`, el contenedor termina justo después de mostrar el mensaje en consola, por lo tanto, su estado es *exited* y no aparece en el listado. Si desea listar contenedores sin importar su estado agregue el parámetro `--all`.

```
$ docker container ls --all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	NAMES
fa668052b884	hello-world	"/hello"	5 seconds ago	Exited (0) 4 sec...	vigilant_bo\hr

Docker mantiene el registro del contenedor aunque haya finalizado, pero si desea hacer limpieza en su ordenador puede utilizar el comando `rm` para eliminar completamente el registro del contenedor.

```
$ docker container rm fa668052b884
```

```
fa668052b884
```

Fíjese que después del comando `rm` se ha incluido el identificador del contenedor (*CONTAINER ID*). Este valor es único para cada contenedor.

En este punto usted es capaz de gestionar imágenes y contenedores. Le recomiendo que utilice unos minutos para ejercitarse con los comandos vistos con otras imágenes que sean de su interés. Estos ejercicios individuales le van a ayudar a fortalecer los conocimientos vistos aprendidos.

## Cambiando el formato en Docker CLI

La mayor parte de los comandos utilizados en Docker devuelven el contenido en forma de tabla. Este formato ayuda mucho cuando tienen que ser listados múltiples valores porque permite resaltar el tipo de contenido a través de columnas. Sin embargo, en ocasiones es necesario cambiar el formato de salida para hacer uso de la información en otros propósitos, por ejemplo:

- Obtener un identificador y pasarlo como parámetro a una función.
- Obtener un listado de nombres separados por un espacio para luego iterar sobre ellos.
- Obtener el resultado en formato JSON para luego modificarlo con la herramienta [JQ<sup>56</sup>](#).

Cualquiera de estas razones, o alguna otra que no haya sido incluida, puede derivar en el deseo de modificar el formato de salida de los comandos en Docker. Para suerte de todos Docker tiene el mismo razonamiento, y para mostrarlo ha incluido el parámetro `--format`, permitiendo realizar modificaciones en el formato de salida.

El parámetro `--format` utiliza la sintaxis de plantilla Golang ([Go Template<sup>57</sup>](#)), la cual permite hacer uso de la estructura `{{ }}` para crear instrucciones adaptadas a sus necesidades. Veamos a continuación algunos ejemplos de cómo utilizar el parámetro `--format`.

### Consultar un valor específico

Si escribe el comando `docker system info` va a obtener muchísima información relacionada con el ordenador que esté utilizando, pero digamos que solamente le interesa saber la CPU disponible, en ese caso puede utilizar el parámetro `--format` de la siguiente forma:

```
$ docker system info --format 'El número de CPU disponible es: {{ .NCPU }}'
```

```
El número de CPU disponible es: 4
```

¡Más directo verdad!

### Obtener el resultado en formato JSON

En el comando anterior se obtuvo el número de CPU disponible utilizando `NCPU` como identificador. Tener a la mano esta información ha sido útil, pero *¿cómo obtener los identificadores disponibles en los comandos?*

Todos los campos brindados por Docker están a su alcance en formato `JSON`, pero luego son mostrados de forma amigable para su fácil comprensión. Realice una pequeña modificación en la salida para obtenerlos nuevamente en formato `JSON`.

---

<sup>56</sup><https://stedolan.github.io/jq/>

<sup>57</sup><https://pkg.go.dev/text/template>

```
$ docker system info --format '{{ json .}}'\n\n{"ID": "...", "Containers": 5, "ContainersRunning": 0, ... ... }
```

Ciertamente ahora está en formato JSON, pero todavía es difícil de ver. Realice una segunda mejora dándole el formato correcto a la salida utilizando la herramienta [jq<sup>58</sup>](#).

```
$ docker system info --format '{{ json .}}' | jq\n\n{\n    "ID": "...",\n    "Containers": 5,\n    "ContainersRunning": 0,\n    ...\n}
```

¡Mucho mejor! Busque el identificador que más le guste e intente obtenerlo de forma individual como mismo se hizo con el primer ejemplo.

## Definir tus propias tablas

Otra manera de cambiar el formato de salida es creando tablas con las columnas que sean de su interés, o dejando el mismo número de columnas pero cambiando el orden en que aparecen. Pongamos un ejemplo, desea listar el último contenedor iniciado en su ordenador, el comando directo es:

```
$ docker container ls --latest\n\nCONTAINER ID IMAGE      ... ...\nb6d1f9e5d1ac hello-world ... ...
```

Luego consulte los identificadores para modificar la salida a su gusto.

---

<sup>58</sup><https://stedolan.github.io/jq/>

```
$ docker container ls --latest --format '{{ json . }}' | jq

{
  "Command": "/hello",
  "CreatedAt": "2022-03-02 15:08:22 +0100 CET",
  "ID": "b6d1f9e5d1ac",
  "Image": "hello-world",
  "Labels": "",
  "LocalVolumes": "0",
  "Mounts": "",
  "Names": "wonderful_raman",
  "Networks": "bridge",
  "Ports": "",
  "RunningFor": "29 seconds ago",
  "Size": "0B (virtual 9.14kB)",
  "State": "exited",
  "Status": "Exited (0) 29 seconds ago"
}
```

Ahora intente crear la salida en tabla utilizando el identificador y el nombre del contenedor solamente.

```
$ docker container ls --latest --format 'table {{.ID}}\t{{.Names}}'

CONTAINER ID NAMES
b6d1f9e5d1ac wonderful_raman
```

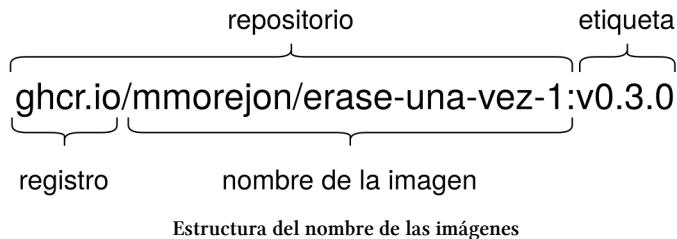
¡Sencillo verdad! Ha sido fácil cambiar el formato de los resultado. De ahora en adelante utilice los diferentes formatos y saque el mayor provecho posible de Docker. En los próximos capítulos se va a hacer uso del formato *json* para mostrar los datos relevantes en cada ejercicio.

## Resumen del capítulo

Durante el capítulo se explicaron dos vías para acceder a la documentación, utilizando los comandos y accediendo a la web oficial. Docker actualiza el sistema múltiples veces al año, por lo tanto, documentarse será importante para no perder la pista.

Se analizó en detalles el flujo de pasos seguido por Docker cuando se pone en funcionamiento un contenedor. Este conocimiento le va a ser útil a la hora de solucionar errores durante el uso de la herramienta.

Se explicó la existencia de múltiples Registros de contenedores para almacenar imágenes. Muchas de las plataformas hoy en día tienen sus propios registros y usted debe tomar la decisión de cuál utilizar.



Por último, se explicaron las diferentes formas que brinda Docker para modificar el formato de salida en los comandos. Esta opción es muy útil si desea combinar la información brindada por Docker con sistemas de infraestructura o con procesos de integración continua.

Con los conocimientos vistos en el capítulo ha comenzado su viaje en el mundo de los contenedores, pero todavía quedan muchos aspectos importantes a incorporar, como por ejemplo:

- *¿Cuál es el comando utilizado al iniciar los contenedores?*
- *¿Será posible modificar este comando en caso de ser necesario?*
- *¿Cómo iniciar múltiples contenedores de forma independiente?*
- *¿Cómo consultar los registros (logs) de los contenedores en funcionamiento?*

Continue leyendo el próximo capítulo para aprender nuevos conocimientos y lograr dar respuesta a estas preguntas.

*¡Mucho ánimo que vas bien!*

# Rutinas con los contenedores

Las actividades realizadas a diario con los contenedores no son diferentes a las que hacemos actualmente con el resto de sistemas, simplemente ha cambiado la forma de hacerlo.

Cuando se desea poner en funcionamiento un sistema, el primer paso es ir al sitio oficial y conocer las instrucciones de su instalación. Luego se descarga el fichero de la aplicación, se instala y se inicia de la forma más fácil posible. Una vez que el sistema está en funcionamiento, se analizan las variantes de configuración y se consultan los registros (*logs*) generados.

En el caso de los contenedores sucede de forma muy similar. Se descarga la imagen que contiene la aplicación y luego se pone en funcionamiento el contenedor de la forma más fácil posible. Luego es necesario conocer las opciones que brinda el sistema para su configuración y los registros generados durante su funcionamiento.

Para lograr realizar estas acciones con los contenedores necesita incluir un nuevo grupo de comandos de Docker. Cada nuevo comando incluye nuevos parámetros, pero no se preocupe, los pasos van a ser dados poco a poco para no perder ningún detalle.

Durante el capítulo se le va a dar respuesta a las siguientes preguntas.

- *¿Cuál es la instrucción que utiliza el contenedor para iniciarse?*
- *¿Se podrá cambiar esta instrucción de inicio del contenedor?*
- *¿Cómo poner en funcionamiento múltiples contenedores?*
- *¿Cómo acceder al contenedor durante su funcionamiento?*
- *¿Cómo consultar los logs generados por los contenedores?*
- *¿Cuántos recursos consume un contenedor en funcionamiento?*

Le recomiendo tener listo su Terminal para ponerse manos a la obra, ya que todos los casos a analizar incluyen ejercicios prácticos.

## Identificar el comando de inicio

Los contenedores son procesos en ejecución y como todo proceso es necesario tener comando como punto de partida.

Docker ha diseñado la estructura de las imágenes para almacenar los ficheros que necesita la aplicación, pero también ha dejado espacio para poner información valiosa como por ejemplo: cómo fue construida, cómo deben ser ejecutados los contenedores, arquitectura donde puede ser ejecutado el contenedor, tamaño de la imagen, entre otros.



Puede utilizar el comando `inspect` para listar todas la información disponible. El ejemplo mostrado continuación utiliza la imagen *hello-world*.

```
$ docker image inspect hello-world --format '{{ json . }}' | jq

{
  "Id": "sha256:18e5af...",
  "RepoTags": [
    "hello-world:latest"
  ],
  "RepoDigests": [
    "hello-world@sha256:4c5f3db4..."
  ],
  "Parent": "",
  ...
}
```

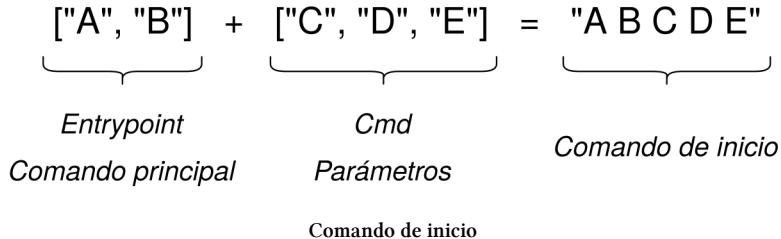


El comando `inspect` necesita tener la imagen descargada en el ordenador. Utilice el comando `image pull` en los casos donde sean necesarios.

En el ejemplo anterior se han utilizado las [técnicas de formato](#) explicadas en el capítulo anterior.

Dentro del grupo de elementos listados existen dos valores con la misión de establecer la instrucción de inicio del contenedor, estos valores son *Entrypoint* y *Cmd*. La tabla que aparece a continuación le va a mostrar el significado de cada uno.

Campo	Tipo	Descripción
<i>Entrypoint</i>	array[string]	Listado de argumentos utilizados como comando en el momento que se inicia el contenedor.
<i>Cmd</i>	array[string]	Parámetros pasados al comando de inicio <i>Entrypoint</i> del contenedor. Si el valor del <i>Entrypoint</i> está vacío, entonces el primer elemento de la lista del <i>Cmd</i> será utilizado como comando de ejecución.



Estos dos valores trabajan de conjunto para establecer la instrucción de inicio en los contenedores, el *Entrypoint* define el comando y el *Cmd* aporta los parámetros. Utilice nuevamente el comando `inspect` para saber cómo ha sido configurada la imagen `hello-world`, en esta ocasión se muestra el subconjunto `Config` porque es el lugar donde están estos valores.

```
$ docker image inspect hello-world --format '{{ json .Config }}' | jq

{
  ...
  "Cmd": [
    "/hello"
  ],
  "Entrypoint": null,
  ...
}
```

En esta ocasión la imagen no tiene un valor asignado en el campo *Entrypoint*, por lo tanto el primer elemento del campo *Cmd* es utilizado como comando de inicio. Todos los contenedores creados con esta imagen van a tener como inicio del binario `hello`. Analice un ejemplo similar, pero cambiando la imagen.

```
$ docker image inspect ghcr.io/mmorejon/erase-una-vez-1:v0.3.0 \
--format '{{ json .Config }}' | jq

{
  ...
  "Cmd": null,
  "Entrypoint": [
    "erase-una-vez-1"
  ],
  ...
}
```

En esta ocasión el campo *Entrypoint* no está vacío, por lo tanto el comando de inicio es la ejecución del binario `erase-una-vez-1`. Este comando no tiene asociados parámetros adicionales porque no existen

valores en el campo *Cmd*. Como puede observar, con tener valores en al menos uno de estos campos el contenedor puede iniciarse sin problemas.

Un tercer ejemplo muy interesante es el comando de inicio de la imagen *Nginx*.

```
$ docker image inspect nginx:1.21.6 --format '{{ json .Config }}' | jq

{
  ...
  "Cmd": [
    "nginx",
    "-g",
    "daemon off;",
  ],
  "Entrypoint": [
    "/docker-entrypoint.sh"
  ],
  ...
}
```

Aquí ambos campos cuentan con información, por lo tanto el resultado es la unión de ambos. Lo primero en ejecutarse es el fichero `docker-entrypoint.sh`, este es un script que realiza algunas sustituciones y configuraciones necesarias para Nginx. Una vez terminado el script se inicia el comando `nginx` utilizando como parámetros `-g daemon off;`.

Como puede percibirse, las alternativas de configuración con estos campos pueden ser variadas. De ahí la importancia de saber inspeccionar las imágenes para conocer su configuración de inicio. Le propongo, ahora que tiene nuevos conocimientos, realizar el mismo análisis del comando de inicio con otras imágenes de su interés. Este ejercicio le permitirá ganar confianza con el comando `inspect`.

## Modificar el comando de inicio

Conocer el comando de inicio permite poner en funcionamiento el contenedor de forma rápida y sencilla, pero no es suficiente, también va a necesitar adaptar este comando a las necesidades del proyecto en caso que hiciese falta.

No piense que por desear cambiar el comando de inicio está haciendo algo incorrecto, todo lo contrario, es el comportamiento esperado. Es muy común tener múltiples entornos de trabajo para un sistema, p.ej: desarrollo, pruebas y producción. En cada entorno varían las configuraciones, por lo tanto, va a ser importante poder modificar el comando inicial en los contenedores. El propósito del cambio es adaptar los sistemas a cualquier situación.

En la sección anterior, se explica que el campo *Cmd* contiene los parámetros adicionales del comando *Entrypoint*, por lo tanto, si desea realizar cambios, el mejor lugar para hacerlo es en *Cmd*. Para llevar a cabo la modificación basta con agregar las nuevas instrucciones al final del comando *Run*, de esta manera quedan reemplazados los valores de la imagen con los elementos puestos en el comando.

```
$ docker container run --help  
Usage: docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Veamos a continuación algunos ejemplos donde se modifica el comando de inicio en diferentes imágenes.

## Imagen ubuntu:22.04

La imagen `ubuntu:22.04` no tiene definido *Entrypoint*, y su *Cmd* es `/bash`. Inicie el contenedor utilizando los valores predeterminados. Una vez ejecutado el comando `bash`, el contenedor terminará sin mostrar nada en el Terminal.

```
$ docker container run ubuntu:22.04  
  
Unable to find image 'ubuntu:22.04' locally  
22.04: Pulling from library/ubuntu  
84df7bf04e98: Pull complete  
Digest: sha256:9fd089c601e4ce6b61bfbba987b63ec6b73eb4ef9f568ad38b58cf0bbb019bff  
Status: Downloaded newer image for ubuntu:22.04
```

Ahora cambie el comando *Cmd*, la nueva orden va a ser listar los elementos presentes en la raíz del sistema. Los nuevos comandos tienen que ser incluidos después del nombre de la imagen.

```
$ docker container run ubuntu:22.04 ls -la /  
  
total 56  
drwxr-xr-x  1 root root 4096 Mar 16 10:05 .  
drwxr-xr-x  1 root root 4096 Mar 16 10:05 ..  
-rwxr-xr-x  1 root root    0 Mar 16 10:05 .dockerenv  
...
```

¡Enhorabuena! Ha cambiado la instrucción de inicio del contenedor.

## Imagen nginx:1.21.6

Esta imagen tiene definido ambos comandos, *Entrypoint* y *Cmd*. Utilizando los valores por defecto va a poder observar el inicio del servidor Nginx.

```
$ docker container run nginx:1.21.6  
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform config\  
uration  
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/  
...
```

Detenga el servicio (*Ctr+C*) y cambie el comando *Cmd* para mostrar la versión de Nginx.

```
$ docker container run nginx:1.21.6 nginx -v  
...  
nginx version: nginx/1.21.6
```

El inicio del comando ha sido similar al anterior. Los mensajes mostrados en pantalla son iguales a los ejemplos anteriores, pero la diferencia aparece al final mostrando la versión de Nginx. Al no cambiar el valor del *Entrypoint*, el arranque sigue siendo el mismo. La modificación viene después, donde se muestra la versión del programa en vez del inicio del servidor web.

## Cambiar el valor del Entrypoint

Las imágenes de los contenedores se crean con un propósito, y por lo general, el comando inicial de ejecución *Entrypoint* no debe ser cambiado. Sin embargo, pudiera darse el caso donde desee cambiar este valor al igual que lo hace con el *Cmd*. Si este fuese el caso debe utilizar el parámetro `--entrypoint` o `-e` para lograrlo.

```
$ docker container run --entrypoint=ls nginx:1.21.6 -la /  
total 76  
drwxr-xr-x  1 root root 4096 Mar 16 11:05 .  
drwxr-xr-x  1 root root 4096 Mar 16 11:05 ..  
-rwxr-xr-x  1 root root    0 Mar 16 11:05 .dockerenv  
drwxr-xr-x  2 root root 4096 Feb 28 00:00 bin  
...
```

En este ejemplo se ha establecido como comando *Entrypoint* `ls` y luego para el *Cmd* se ha pasado como parámetro `-la /`. Como resultado final se han listado los elementos del directorio raíz.

Como puede apreciar se puede cambiar tanto el *Entrypoint* como el *Cmd*, pero le recomiendo hacer uso del *Cmd* siempre que sea posible.

## Iniciar contenedores de forma interactiva

Acceder a las máquinas virtuales es una actividad común en los entornos tecnológicos. Se accede a ellas para instalar sistemas, realizar configuraciones o simplemente para comprobar el estado de un proceso. Si necesita probar un nuevo sistema operativo, la solución pasa por crear una nueva máquina virtual con la nueva versión para luego acceder a ella. De una u otra forma, hay momentos donde es necesario acceder al sistema.

Los contenedores conocen esta necesidad, por eso se ha habilitado la opción de acceder a ellos (*modo interactivo*) para ejecutar los comandos. Para utilizar el modo interactivo tiene que hacer uso de dos nuevos parámetros.

- `--interactive` o `-i`: permite mantener abierta la entrada de comandos STDIN.
- `--tty` o `-t`: permite interactuar con la consola a través de comandos.

Ahora que conoce los nuevos parámetros, utilice la imagen de *Ubuntu Server* para acceder de forma interactiva.

```
$ docker container run -it ubuntu:22.04
```

```
root@6fe8bdfb8353:/#
```

Una vez dentro del contenedor ejecute los comandos que desee, por ejemplo:

```
$ cat /etc/os-release
$ apt update
```

No tenga miedo de romper o eliminar ficheros dentro del contenedor, recuerde que si algo va mal siempre puede salir con el comando *Exit* y empezar de nuevo.

Un detalle importante a tener en cuenta en el modo interactivo, solo puede ser utilizado con los comandos `bash`, `sh` u otro similar. Si utiliza un comando diferente a los mencionados no podrá acceder al contenedor. En el ejemplo realizado con la imagen *Ubuntu Server* ha funcionado porque la instrucción existente en *Cmd* es `bash`.

Para poner en práctica lo mencionado anteriormente realice el siguiente comando. Fíjese que el valor *Cmd* ha sido modificado para establecer `ls` en lugar de `bash`. El resultado va a ser listar las carpetas, pero sin acceder al contenedor.

```
$ docker container run -it ubuntu:22.04 ls  
bin dev home media      opt root sbin sys usr  
boot etc lib          mnt      proc run srv tmp var
```

Realice un ejercicio similar con la imagen Nginx, esta vez se ha modificado el *Cmd* para establecer el comando `bash`.

```
$ docker container run -it nginx:1.21.6 bash  
root@faf47b73cac1:/#
```

También puede intentar cambiar `sh` por `bash` y el resultado va a ser similar.

```
$ docker container run -it nginx:1.21.6 sh
```

## Iniciar múltiples servicios independientes

Poner en funcionamiento un contenedor es un gran paso, pero iniciar múltiples contenedores de forma independiente es mucho mejor.

Hasta el momento, todos los contenedores iniciados han estado anclados al control del Terminal. Esto está bien en un inicio, pero no puede desplegar cientos de ellos con esta filosofía. En los entornos reales, los contenedores se inician como procesos independientes, luego se obtienen los registros generados para saber lo que está sucediendo internamente.

Para lograr este comportamiento Docker utiliza el parámetro `--detach` o `-d` del comando *Run*. Este parámetro inicia el contenedor en un hilo independiente a la consola, y devuelve como referencia el identificador del contenedor. Luego puede utilizarse este identificador para conocer el estado del proceso, recolectar *logs* o detenerlo si es preciso.

Preste atención al siguiente comando donde se utiliza el nuevo parámetro.

```
$ docker container run --detach ghcr.io/mmorejon/erase-una-vez-1:v0.3.0  
4fce1b15136d715d7499aaa7a0ad9318f7fd4439d6486924549828f704d2c474
```

Después de ejecutar el comando vuelve a tener control sobre su Terminal. Puede realizar nuevas tareas con total tranquilidad sin afectar su servicio. Intente listar todos los contenedores en ejecución como parte del ejemplo, en el resultado va a aparecer el contenedor iniciado previamente.

```
$ docker container ls
```

Va a llegar el momento donde tenga muchos contenedores en el listado, cada uno funcionando de forma independiente, pero *¿qué ha pasado con los registros (logs) que se veían en el Terminal? ¿Cómo se pueden ver los registros para cada contenedor?*

Avance a la próxima sección para dar respuesta a estas preguntas sobre los *logs* en los contenedores.

## Consultar los logs

Todos los sistemas brindan visibilidad de las acciones realizadas a través de mensajes de salida. La correcta gestión de estos registros es importante porque esta información es utilizada luego para mejorar el rendimiento y la estabilidad de los sistemas. Las formas que existen de gestionar estos mensajes son diversas dependiendo del entorno donde se encuentre, pero las dos variantes más comunes son: almacenarlos en un fichero y/o imprimirlas directamente en la consola.

Cuando las aplicaciones son desplegadas con Docker existe una forma sencilla de conocer los mensajes generados, utilizando el comando `logs`. Este comando está asociado a los contenedores y no a las imágenes, porque son los contenedores los que constituyen un proceso en ejecución.

El comando `logs` tiene múltiples parámetros de configuración. En breve se van a analizar los más importantes, pero no se limite, utilice la documentación oficial para ir aún más lejos.

A través del parámetro `--help` va a poder conocer las opciones disponibles.

```
$ docker container logs --help
```

```
Usage: docker container logs [OPTIONS] CONTAINER
...
```

Para analizar los *logs* de una aplicación lo primero es tener la aplicación, ¿cierto? Puede hacer uso del contenedor iniciado anteriormente, pero si no lo tiene disponible inicie otro con el siguiente comando.

```
$ docker container run --detach ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
```

```
cd709d41f9ac0415010ce803a61c36cd439064f94ff0c09d23128482a7b94900
```

Una vez creado el contenedor puede pasar a consultar los registros. El único parámetro obligatorio en el comando es el identificador del contenedor.

```
$ docker container logs --follow cd709d41f9ac  
  
hostname: cd709d41f9ac - érase una vez ...  
hostname: cd709d41f9ac - érase una vez ...  
hostname: cd709d41f9ac - érase una vez ...
```

Como resultado va a obtener los mensajes generados por la aplicación. El mensaje es la unión del *hostname* del contenedor junto con el texto *erase-una-vez*. El valor del hostname va a coincidir con el identificador del contenedor.

También note el uso del parámetro `--follow`, este valor le va a permitir quedarse a la espera de los próximos mensajes que genere el contenedor. Pruebe ejecutar el comando `logs` sin utilizar este parámetro para notar la diferencia entre ambos. Para salir del control de la consola utilice la combinación de teclas *Ctr+C*.

El resto de parámetros disponibles para el comando `logs` permiten hacer algunos trucos adicionales, como por ejemplo:

- `-since`: muestra los registros a partir de una fecha (p.ej 2013-01-02T13:23:37Z).
- `-tail`: número de líneas a mostrar desde el final.
- `-until`: muestra los registros previos a una fecha (p.ej 2013-01-02T13:23:37Z).

Anímese a utilizar alguno de estos nuevos parámetros y compare los resultados. Recuerde, la curiosidad es la mejor compañía durante el estudio de nuevas tecnologías.

## Conocer el consumo de recursos

Todos los sistemas consumen recursos de una u otra forma. Recursos como CPU, RAM y Disco son algunos de los más comunes en todas las plataformas, sin embargo, la mala gestión de estos podría costar mucho dinero a su proyecto.

El primer paso para controlar los recursos de los contenedores es conocer su consumo. Para acceder a esta información se utiliza el comando `stats`. El siguiente ejemplo muestra cómo hacer uso del nuevo comando. Como requisito previo tiene que tener al menos un contenedor activo, porque los datos mostrados se obtienen en tiempo real.

```
$ docker container stats \
--no-stream \
--format 'table {{.ID}}\t{{.Name}}\t{{.CPUPerc}}\t{{.MemPerc}}'

CONTAINER ID  NAME          CPU %     MEM %
ff4cf2ac9fb4  kind_chatelet  0.01%    0.07%
b994c5d21b79  competent_keller 0.03%    0.14%
```

El parámetro `--no-stream` recopila la información de los contenedores y los muestra en pantalla. A modo de prueba ejecute el comando sin este parámetro para recibir los datos en tiempo real. El segundo parámetro utilizado ya es conocido, se utiliza para mostrar aquellos elementos que se quieren resaltar. Elimine este último parámetro si desea conocer todos los datos brindados por el comando `stats`.

```
$ docker container stats
```

Fíjese en los detalles, casi todos los recursos muestran el valor de consumo actual, pero también el límite que pueden llegar a alcanzar. Ambos datos son muy importantes, en ocasiones los sistemas fallan por la falta de recursos en la máquina, tener ambos valores le permite identificar el problema y encontrar una solución de forma rápida. Tenga esto presente cuando utilice los contenedores en entornos reales.

Con el comando `stats` puede conocer el consumo de la CPU y la RAM, pero falta algo más, el espacio en disco. Este último recurso se puede mostrar con el comando `df`, perteneciente al grupo `system`.

```
$ docker system df
```

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	47	3	15.31GB	15.1GB (98%)
Containers	22	3	31.61MB	31.61MB (100%)
Local Volumes	13	0	315.4MB	315.4MB (100%)
Build Cache	104	0	468B	468B

Como puede observar, aparece el espacio utilizado agrupado por conceptos. En este momento le es familiar *Images* y *Containers*, pero los demás van a ser incluidos en próximos capítulos.

Analice la columna *RECLAIMABLE*, los valores mostrados indican el espacio que posible de recuperar si se realizan acciones de mantenimiento o limpieza. Los comandos de mantenimiento en Docker son de gran importancia para liberar espacio y evitar saturaciones con el almacenamiento.

Con la ayuda de los comandos mostrados va a poder monitorizar los recursos utilizados por los contenedores, utilizar solamente aquello que necesitas es una virtud.

## Ejercicios para practicar lo aprendido

Después de haber incorporado nuevos conceptos y comandos es momento de practicar. Los ejercicios que aparecen a continuación tienen la intención de pasar un rato agradable delante del Terminal.

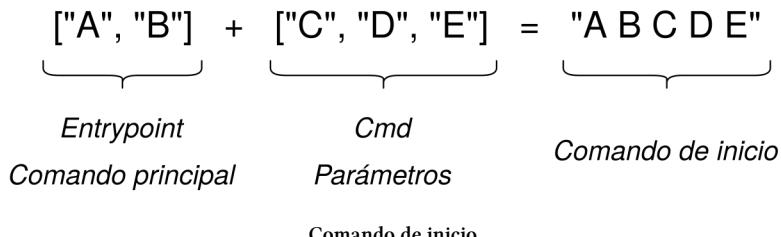
1. Identifique cuál es el comando de inicio en las siguientes imágenes.
  - alpine:3.15.1
  - postgres:14.2
  - golang:1.18.0
2. Muestre el comando de inicio utilizando la estructura que aparece a continuación. Sustituya los valores con formato de código por la información de las imágenes del ejercicio anterior.

El comando de inicio de la imagen nombre:etiqueta es: Entrypoint Cmd.
3. Inicie de forma independiente las siguientes imágenes de contenedores. Una vez iniciados consulte los *logs* de cada uno para asegurarse que el funcionamiento es correcto.
  - postgres:14.2
  - mysql:8.0.28
4. Consulte la CPU y RAM consumida por los contenedores iniciados en el ejercicio anterior. Identifique cuál de los dos servicios utiliza más memoria.
5. Elimine todos los contenedores en funcionamiento de su ordenador. Intente realizar esta operación en una sola línea de comandos.

## Resumen del capítulo

Durante el capítulo se mostraron los comandos utilizados con mayor frecuencia en la actividad diaria con Docker.

El primer paso fue conocer la instrucción de inicio a través del comando `inspect`. La instrucción de inicio es la unión de los elementos *Entrypoint* y *Cmd*. Conocer estos valores le permite entender el funcionamiento del sistema, pero mejor aún, va a ser capaz de modificarlos para adaptar la aplicación al entorno de ejecución.



También se explicó la función del parámetro `--detach` del comando `run`. Iniciar los contenedores en segundo plano (*background*) le brinda independencia al servicio durante su funcionamiento. Es la configuración recomendada para todo los contenedores que desee iniciar.

Dar seguimiento al estado de los servicios es vital para detectar errores de forma temprana. En este punto su mejor aliado es el comando `logs`, brindando un forma sencilla y rápida para visualizar todos los mensajes generados por los contenedores. Por último, pero no menos importante, se mostraron los comandos `stats` y `system df` para conocer los recursos utilizados por los contenedores.

Los comandos vistos durante el capítulos constituyen las rutinas más repetidas al trabajar con contenedores. Le recomiendo dedicar tiempo para practicarlas con diferentes imágenes.

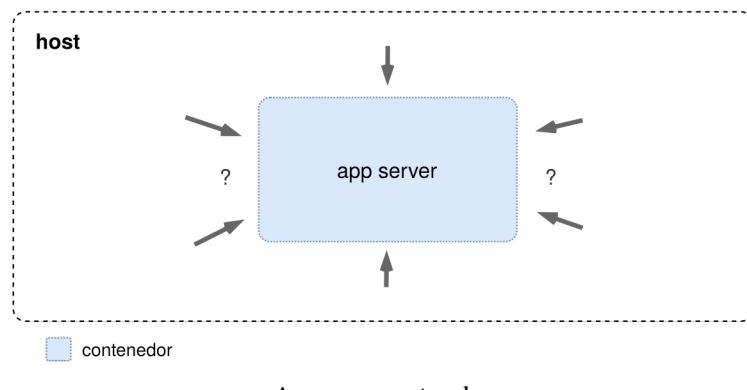
En el próximo capítulo conocerá sobre el funcionamiento de las redes en Docker y cómo acceder a sus aplicaciones a través de la gestión de puertos.

*¡Mucho ánimo y siga adelante!*

# Publicar y consumir servicios

La comunicación y el acceso entre contenedores son piezas fundamentales en Docker. Garantizar el aislamiento de los procesos a través de las redes es un requisito si está interesado en gestionar correctamente los servicios.

El primer paso fue iniciar múltiples contenedores independientes, pero el siguiente es poder acceder a ellos. Esta comunicación puede establecerse a través del navegador, pero también desde otro contenedor iniciado en la misma máquina. En ambos casos, va a necesitar los conocimientos en redes de Docker para poder llevarlo a cabo correctamente.



En las próximas secciones se va a mostrar la configuración establecida por Docker durante su instalación. Luego se van a explorar nuevos comandos para gestionar las redes y las direcciones IPs asignadas a los contenedores.

También conocerá variantes para exponer o publicar los servicios de los contenedores. Dicho en otras palabras, va a poder iniciar un servidor Nginx o una base de datos MySQL para que otras aplicaciones accedan a estos servicios con facilidad.

Si se encuentra listo. ¡Empezamos!

## Estructura de redes en Docker

Docker configura de tres redes durante el proceso de instalación, cada una utilizando un controlador (*driver*) distinto. Los controladores (*drivers*) de redes son subsistemas con la capacidad de ser añadidos o eliminados (*plug-in*) de forma fácil, y cada uno brinda un comportamiento distinto en la comunicación de los contenedores.

Utilice el gestor de comandos `network` para listar las redes creadas de forma predeterminada. Si en su listado aparecen más de tres elementos significa que ha utilizado esta funcionalidad en otro momento, o ha sido creado por alguna otra aplicación que utilice las redes de Docker.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
41b63112aceb	bridge	bridge	local
0f11289d83d9	host	host	local
34a3cdbfa391	none	null	local

Los controladores `bridge`, `host` y `null` los instala Docker. Cada uno tiene un identificador similar a los identificadores de los contenedores. Antes de pasar a la descripción de los controladores, le propongo dar una mirada a los comandos disponibles para gestionar redes.

```
$ docker network --help
```

#### Manage networks

##### Commands:

connect	Connect a container to a network
create	Create a network
disconnect	Disconnect a container from a network
inspect	Display detailed information on one or more networks
ls	List networks
prune	Remove all unused networks
rm	Remove one or more networks

## Controlador bridge

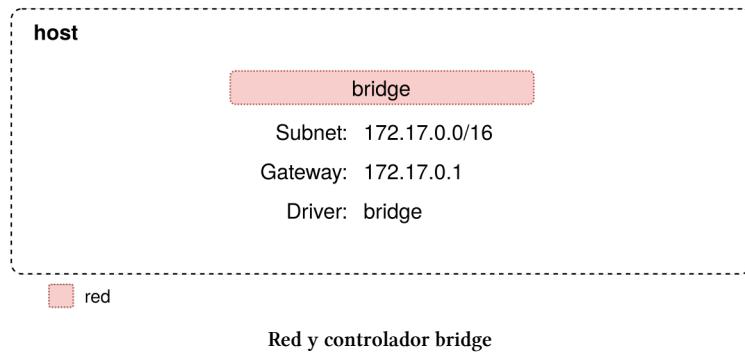
Este es el controlador más utilizado por los servicios, de hecho es el controlador por defecto. La traducción del nombre es puente, y su funcionamiento consiste en crear un segmento de red para aislar los contenedores del resto de servicios. La forma rápida de mostrar la configuración de este segmento de red es utilizando el comando `inspect`.

```
$ docker network inspect bridge
```

En la sección IPAM (*IP Address Management*) se muestra el rango de *IPs* reservadas. Todos los contenedores iniciados con este controlador van a obtener una IP en este rango; a no ser que le indique una configuración diferente.

```
$ docker network inspect bridge --format='{{ json .IPAM }}' | jq

{
  "Driver": "default",
  "Options": null,
  "Config": [
    {
      "Subnet": "172.17.0.0/16",
      "Gateway": "172.17.0.1"
    }
  ]
}
```



Red y controlador bridge

Este controlador va a ser utilizado en el resto de secciones para crear redes y conectar contenedores.

## Controlador host

El controlador *host* no aísla el contenedor, y tampoco le asigna una dirección en un segmento de red. Este controlador comparte el espacio de red de la máquina que lo ha iniciado. Esto significa que si el contenedor acepta peticiones por el puerto 80, entonces la máquina donde este el contenedor, también aceptará peticiones en el mismo puerto.

Existen casos donde le interese utilizar este controlador, por ejemplo, si necesita dar una prioridad al rendimiento del sistema, o si necesita hacer uso de un gran número de puertos para el servicio. Tenga especial cuidado con el solapamiento de puertos si inicia múltiples contenedores con este controlador.

## Controlador null

El controlador *null* se utiliza para eliminar todas las configuraciones de red del contenedor. Así de sencillo, no va a existir comunicación salvo el acceso al contenedor desde la máquina.

## Conocer la IP del contenedor

Docker le asigna una *IP* a los contenedores iniciados en redes con el controlador *bridge*. La *IP* es la dirección del contenedor cuando otro servicio desee comunicarse con él.

Para obtener la *IP* de un contenedor se va a utilizar la aplicación [erase-una-vez-2<sup>59</sup>](#). Esta aplicación permite ser iniciada en modo cliente / servidor. El modo servidor, establecido por defecto, permite la entrada de peticiones en el *endpoint /echo*. El resultado de la petición es el nombre de host (*ID del contenedor*) y el texto *érase una vez ....*

```
$ docker container run --detach ghcr.io/mmorejon/erase-una-vez-2:v0.3.0
```

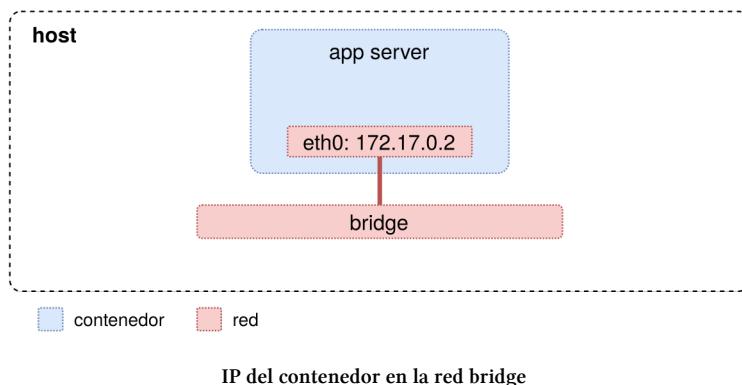
```
1d91d577fd22583a3598f16ea48fc6ea39bc2406747df74abb5a960c371aa5de
```

Como no se ha indicado ningún parámetro de red, el contenedor se ha iniciado en la red por defecto, *bridge*. Para conocer la *IP* asignada al contenedor utilice el comando *inspect* y reduzca el resultado al campo *IPAddress*.

```
$ CONTAINER_ID=$(docker container ls --last 1 --filter status=running --quiet)
$ docker container inspect $CONTAINER_ID \
    --format '{{ json .NetworkSettings.Networks.bridge.IPAddress }}'

"172.17.0.2"
```

En esta ocasión se han utilizado dos comandos, el primero obtiene el identificador del último contenedor iniciado y luego se utiliza este identificador para inspeccionar el contenedor. El resultado de ambos comandos es la dirección *IP* del contenedor: 172.17.0.2.



<sup>59</sup><https://github.com/mmorejon/erase-una-vez-2>

## Conocer los contenedores conectados a una red

Existe otra manera para conocer la dirección *IP* de un contenedor, listando los contenedores conectados a una red, en este caso la red `bridge`. Utilice el comando `inspect` sobre la red `bridge` y filtre la salida con el parámetro `--format` para obtener el listado de contenedores.

```
$ docker network inspect bridge --format='{{ json .Containers }}' | jq

{
  "1d91d577fd22583a3598f16ea48fc6ea39bc2406747df74abb5a960c371aa5de": {
    "Name": "zen_darwin",
    "EndpointID": "ebcb88cc73373b28fb62fdf7928fb64579043d52734f56fe4c9bf4474771cd7",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
}
```

En el listado aparece un contenedor conectado a la red `bridge`. El identificador de este contenedor va a corresponder con la aplicación iniciada en modo servidor. En el campo `IPv4Address` va a encontrar la *IP* buscada.

## Comunicación entre contenedores

Los contenedores necesitan comunicarse entre ellos para intercambiar información. Este es el comportamiento natural de ellos en los entornos de producción. Para comunicar múltiples contenedores se necesitan al menos dos de ellos, por lo tanto, utilice el contenedor iniciado anteriormente en modo servidor, pero también inicie un segundo en modo cliente.

Para activar al modo cliente en la aplicación `erase-una-vez-260` cambie el valor del parámetro `entrypoint`. De forma adicional se ha utilizado el parámetro `env` para establecer dos variables de entorno, con ellas se le indica al cliente cómo acceder al servidor. Sustituya la *IP* del ejemplo por la de su servidor.

---

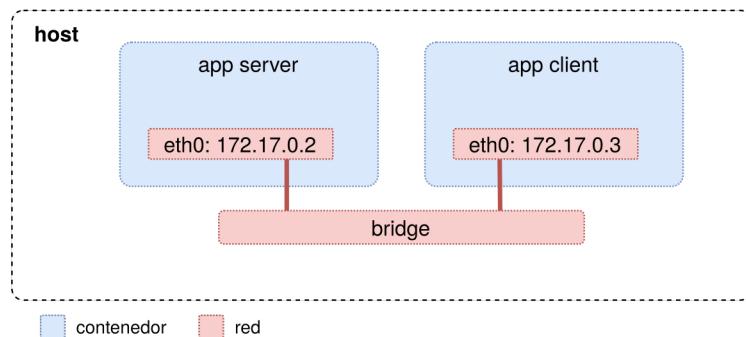
<sup>60</sup><https://github.com/mmorejon/erase-una-vez-2>

```
$ docker container run \
--entrypoint client \
--env SERVER_URL=http://172.17.0.2:8000 \
--env ENDPOINT=/echo \
ghcr.io/mmorejon/erase-una-vez-2:v0.3.0

{
  "hostname": "1d91d577fd22",
  "message": "érase una vez ..."
}
```

Utilice *Ctr+C* para detener y salir del contenedor.

El resultado de haber iniciado la aplicación en modo cliente es un mensaje con el nombre del *host* y el mensaje “*érase una vez ...*”. Compruebe que el nombre del *host* corresponde con el identificador del contenedor en modo servidor.



Contenedores en la misma red

El resultado ha sido un éxito, ambos contenedores se comunicaron correctamente después de haber sido iniciados en la red *bridge*, que es la configuración por defecto.

## Comunicación desde diferentes redes

*¿Qué pasaría si se crea una nueva red? ¿Pueden comunicarse los contenedores desde redes distintas?*

La respuesta es no. Los contenedores no pueden comunicarse desde distintas redes. Docker garantiza el aislamiento entre redes todo momento, por lo tanto, si crea una nueva red, se crea un nuevo segmento para la asignación de direcciones *IPs*, así como también se aplican reglas para aislar las aplicaciones desplegadas en esta red. Los pasos que aparecen a continuación ponen en práctica esta afirmación.

Adicione una red utilizando el controlador *bridge*. La nueva red se va a llamar *second*.

```
$ docker network create --driver bridge second  
5d56bd8ad9f9afa69a8b8b0718457bd2527e3e28d23a92e7dcd9fb0e9ee77c4d
```

Como ya es de costumbre, el resultado en pantalla es el identificador de la nueva red. Ahora liste las redes para visualizar el elemento creado. Aproveche también para conocer el nuevo segmento de red asignado.

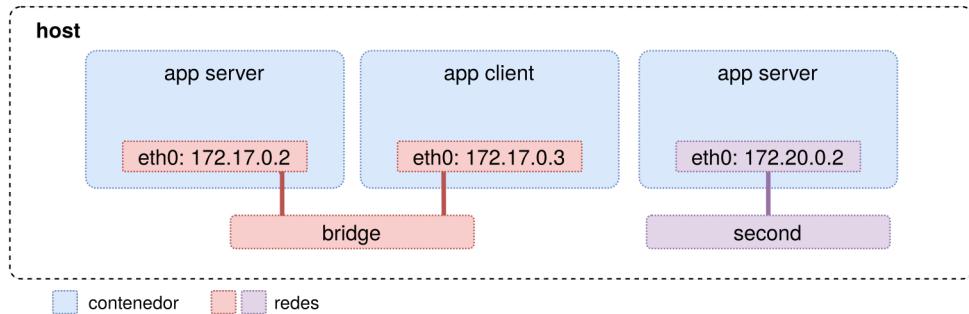
```
$ docker network inspect second --format='{{ json .IPAM }}' | jq  
  
{  
    "Driver": "default",  
    "Options": {},  
    "Config": [  
        {  
            "Subnet": "172.20.0.0/16",  
            "Gateway": "172.20.0.1"  
        }  
    ]  
}
```

El próximo paso es iniciar la aplicación en modo cliente, pero esta vez indique el uso red *second*. La asignación de un contenedor a una red se realiza con el parámetro `--network`. Fíjese que el resto de parámetros han permanecido iguales. Recuerde cambiar la IP del servidor según sea el caso.

```
$ docker container run \  
  --entrypoint client \  
  --env SERVER_URL=http://172.17.0.2:8000 \  
  --env ENDPOINT=/echo \  
  --network second \  
  ghcr.io/mmorejon/erase-una-vez-2:v0.3.0
```

```
2022/03/27 12:43:27 Get "http://172.17.0.2:8000/echo": dial tcp 172.17.0.2:8000: i/o time\  
out
```

Las sospechas han sido confirmadas, la aplicación no ha podido realizar la petición al servidor. El mensaje nos confirma el aislamiento existente entre las redes en Docker, desde el punto de vista de seguridad, es un elemento bueno contar con este comportamiento.



### Aislamiento entre redes

Le invito a realizar un par de pasos más. Inicie otra instancia de la aplicación en modo servidor en la red *second*. Luego obtenga la IP del nuevo servidor y vuelva a intentar la comunicación entre ambos contenedores.

```
$ docker container run \
--detach \
--network second \
ghcr.io/mmorejon/erase-una-vez-2:v0.3.0

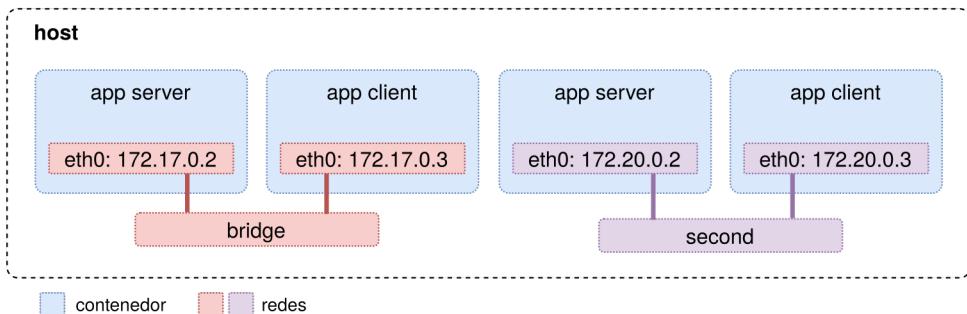
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container inspect $CONTAINER_ID \
--format '{{ json .NetworkSettings.Networks.second.IPAddress }}'

"172.20.0.2"

$ docker container run \
--entrypoint client \
--env SERVER_URL=http://172.20.0.2:8000 \
--env ENDPOINT=/echo \
--network second \
ghcr.io/mmorejon/erase-una-vez-2:v0.3.0

{
  "hostname": "21b6ec53bfa9",
  "message": "érase una vez ..."
}
```

En este segundo intento han podido conectarse los servicios, en este caso el valor del *hostname* ha cambiado, ahora corresponde con el último servicio desplegado.



Múltiples redes con múltiples contenedores

## Publicar los puertos en la máquina

Conocer cómo se comunican los contenedores a través de las redes internas es un gran avance, pero también es necesario permitir el acceso desde el exterior a los contenedores. Imagine el siguiente caso, desea publicar su sitio web utilizando el servidor Nginx, *¿qué opciones brinda Docker para habilitar el acceso a este servicio desde el navegador web?*

Docker permite exponer los puertos del contenedor a través del parámetro *Publish*. Este parámetro asocia dos puertos, uno en la máquina y otro en el contenedor utilizando reglas internas. Inicie un nuevo contenedor e identifique los nuevos parámetros utilizados en el comando:

```
$ docker container run \
--detach \
--publish 9000:8000 \
ghcr.io/mmorejon/erase-una-vez-2:v0.3.0
```

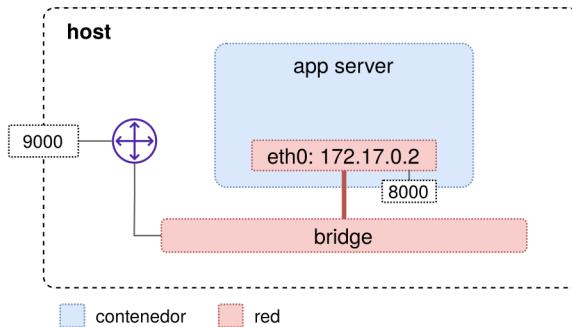
0710c5346fd708f6bbaaf456b7fde1f76bb3aba25801e4e32aac3a66e8f7d826

Acceda a su navegador web al enlace <http://localhost:9000/echo> o utilice el comando *curl* en el mismo propósito, en ambos casos el resultado va a ser el mismo.

```
$ curl http://localhost:9000/echo
```

```
{
  "hostname": "a8d2de470e65",
  "message": "érase una vez ..."
}
```

Seguramente se imagina lo sucedido, pero de todas formas demos algunos detalles. El comando *Publish* ha dirigido el tráfico desde puerto 9000 de su máquina, hacia el puerto 8000 del contenedor. El puerto 8000 recibe la petición y envía de regreso el mensaje.



#### Publicar el puerto del contenedor

Es muy sencillo exponer los puertos de un contenedor, con adicionar el parámetro *Publish* ha logrado su objetivo. Este parámetro puede incluirlo en el comando múltiples veces dependiendo del número de puertos que quiera exponer, si desea exponer tres puertos, debe incluirlo tres veces.

Otro detalle a tener en cuenta es el solapamiento de puertos. Usted es quien indica el puerto del *host*, por lo tanto, si intenta iniciar dos contenedores en el mismo puerto recibirá un error. El mensaje de error va a mencionar que el puerto está en uso.

La relación entre los puertos del *host* y del contenedor pueden verse al listar los contenedores en ejecución, aparecen en la columna *PORTS*.

```
$ docker container ls --format 'table {{.ID}}\t{{.Image}}\t{{.Ports}}'
```

CONTAINER ID	IMAGE	PORTS
0710c5346fd7	ghcr.io/mmorejon/erase-una-vez-2:v0.3.0	0.0.0.0:9000->8000/tcp

Otra variante es utilizar el comando `inspect` sobre el contenedor.

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container inspect $CONTAINER_ID \
--format '{{ json .NetworkSettings.Ports }}' | jq

{
  "8000/tcp": [
    {
      "HostIp": "0.0.0.0",
      "HostPort": "9000"
    }
  ]
}
```

Realice un ejemplo más, esta vez utilice la imagen del servidor Nginx. Luego acceda al siguiente enlace en su navegador: <http://localhost:8080> para comprobar el correcto funcionamiento del servidor web.

```
$ docker container run \
--detach \
--publish 8080:80 \
nginx:1.21.6
```

Con los ejemplos analizados hasta el momento tiene garantizado una gran variedad de casos de uso de la industria para llevar a la práctica. De forma individual le recomiendo iniciar otros servicios utilizando el parámetro *Publish*. La práctica le va a ayudar a fijar los conocimientos.

## Ejercicios para practicar lo aprendido

Después de haber incorporado nuevos conceptos y comandos es momento de practicar. Los ejercicios que aparecen a continuación tienen la intención de pasar un rato agradable delante del Terminal.

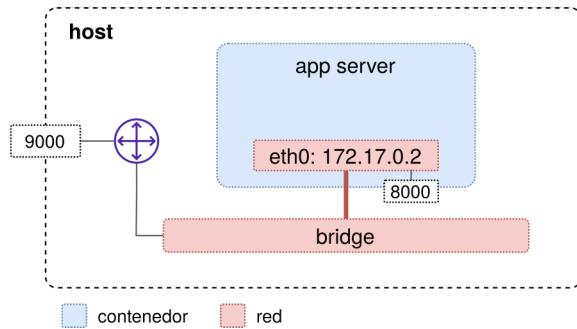
1. Agregue dos redes a la lista de Docker con los nombres: `services` y `databases`.
2. Muestre en pantalla los segmentos de redes asignados a cada una de las redes creadas.
3. Inicie un contenedor en la red `databases` con la imagen `mongo:5.0.6`. Luego inicie otro contenedor utilizando la imagen `mongo:5.0.6` en modo cliente y realice una conexión de prueba con la base de datos MongoDB.
4. Inicie un contenedor en la red `services` con la imagen `mongo:5.0.6` y publicando el puerto `27017` con la máquina local. Luego instale en su ordenador el cliente de MongoDB y realice una conexión utilizando la dirección del servidor `localhost:27017`.

## Resumen del capítulo

Durante el capítulo se han dado los primeros pasos en la temática redes con contenedores. Este es un mundo fascinante y muy amplio, por lo que se han seleccionado aquellos aspectos fundamentales para su inicio. Le animo a que siga profundizando por su cuenta y dele un vistazo a la documentación oficial.

Al comienzo del capítulo se explicaron qué son las redes para Docker y cuáles eran las características de los controladores de redes instalados por defecto (*drivers*). El controlador más utilizado es *bridge*, por tal motivo fue el seleccionado para realizar los ejemplos.

Las redes en Docker pueden utilizarse para aislar los servicios, brindando seguridad y confianza cuando necesite desplegar múltiples servicios en un mismo *host*. Por último, pero no menos importante, se ha utilizado parámetro *Publish* para exponer los puertos del contenedor en el *host*.



Publicar el puerto del contenedor

*¡Enhorabuena por terminar otro capítulo!*, pero todavía quedan retos igual de interesantes. En el próximo capítulo se explica cómo gestionar la persistencia de la información con los contenedores.

# Configuraciones y persistencia de datos

Las configuraciones y la persistencia de la información juegan un papel importante en las aplicaciones. Ahora que ha dado el salto al mundo de contenedores se hace necesario identificar cómo Docker gestiona ambos elementos.

La mayoría de sistemas permiten modificar su comportamiento a través de configuraciones. Estas configuraciones pueden ser encontradas en ficheros para facilitar su acceso y gestión, seguramente ha accedido a alguno de ellos para cambiar los límites de memoria o para habilitar alguna funcionalidad. La buena noticia es que estos ficheros van a seguir existiendo en el mundo de los contenedores, pero va a cambiar la forma de gestionarlos. En este capítulo va a aprender las variantes que existen según sea el caso de uso.

De igual forma, la persistencia de los datos implica un nivel más en la gestión de contenedores. Muchos de los sistemas que conoce almacenan información, y este almacenamiento lo realizan mayormente en estructuras de carpetas y ficheros en la máquina donde se ejecutan. Un ejemplo de estos sistemas son las base de datos.

Las dos problemáticas mencionadas encontrarán respuestas a lo largo del capítulo, así como también podrá responder las siguientes preguntas:

- *¿Qué son los volúmenes de Docker?*
- *¿Cómo adaptar los contenedores a múltiples entornos de desarrollo?*
- *¿Cómo gestionar aplicaciones con persistencia de datos?*

El almacenamiento en los sistemas es crítico por el valor de los datos hoy en día. Siga adelante con el capítulo y descubra cómo abordar esta temática a través de los contenedores.

## ¿Por qué pierdo los datos?

Todos los datos y configuraciones generados por el contenedor se almacenan en su estructura de ficheros. Al ser generados por el contenedor significa que no forman parte de los ficheros que existen en la imagen, por ende, si reinicia el contenedor pierde todos estos nuevos ficheros. Tenga presente lo siguiente, cada vez que inicie un contenedor, el punto de inicio es la estructura de ficheros existente en la imagen.

No sería usted el primero en llevarse las manos a la cabeza, al ver que ha perdido toda la información de su base de datos después de detener su contenedor. Veamos esta problemática a través de un

ejemplo con Redis. Redis<sup>61</sup> es una base de datos *open-source* que permite almacenar combinaciones de llave/valor de forma rápida y sencilla. La imagen oficial de Redis está en Docker Hub<sup>62</sup>.

El primer paso es iniciar Redis dentro de una red con el mismo nombre.

```
$ docker network create \
  --driver bridge \
  redis

c0fa54ccf4bc5a2c58a76f1e82a2039ebb17ebffedd99975db58806f6005e772

$ docker container run \
  --detach \
  --network redis \
  redis:6.2.6

52cf513250907103c4df2b4688253ac102a35cd87ed0809f98d3710606e05d4e
```

Liste los contenedores y compruebe que Redis se ha iniciado correctamente. También puede consultar los registros del contenedor para comprobar que está listo para aceptar peticiones.

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container logs $CONTAINER_ID

1:M 02 Apr 2022 22:59:57.302 * Running mode=standalone, port=6379.
1:M 02 Apr 2022 22:59:57.302 # Server initialized
1:M 02 Apr 2022 22:59:57.303 * Ready to accept connections
```

El próximo paso es iniciar un segundo contenedor de Redis, en la misma red, pero en modo el cliente. Desde este contenedor se van a realizar las peticiones para almacenar una llave en el servidor.

```
$ docker container run \
  --network redis \
  -it \
  redis:6.2.6 redis-cli -h $CONTAINER_ID

$ 52cf51325090:6379> set mykey somevalue
OK
$ 52cf51325090:6379> get mykey
"somevalue"
```

---

<sup>61</sup><https://redis.io/>

<sup>62</sup>[https://hub.docker.com/\\_/redis](https://hub.docker.com/_/redis)

El `CONTAINER_ID` de Redis ha sido utilizado para identificar al contenedor dentro de la red `redis`, otra variante es utilizar la `IP` del contenedor. La capacidad de utilizar el ID del contenedor como alias dentro de la red existe para las redes creadas por los usuarios. Si hubiese utilizado la red por defecto `bridge` solamente hubiese podido acceder utilizando la dirección `IP`.

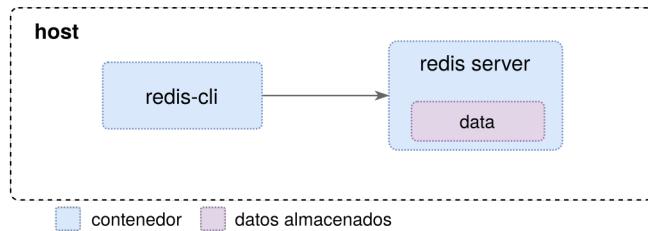


Diagrama del servicio de Redis

Utilice la combinación `Ctr-C` para salir del cliente Redis, luego vuelva a acceder y compruebe que todavía existe el valor `somevalue` asociado a la llave `mykey`.

```
$ 52cf51325090:6379> get mykey
```

```
"somevalue"
```

## El problema viene justo ahora

Si por alguna razón tiene que detener el contenedor, ya sea por mantenimiento o por cualquier otro motivo, va a perder toda la información almacenada. Detenga el contenedor actual y cree otro en su lugar. Luego acceda a través del cliente y compruebe que ha perdido el valor de la llave.

```
$ docker container stop $CONTAINER_ID
```

```
52cf51325090
```

```
$ docker container run \
--detach \
--network redis \
redis:6.2.6
```

```
4e529f9ba930b0e6cfb1f4cf7474a7edfff8b1ae2dc704e9e5bbad6fac61527a
```

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID
```

```
$ 4e529f9ba930:6379> get mykey  
(nil)
```

Como bien muestra la última consulta, sus datos se han perdido. Para este simple ejemplo pudiese parecer un error sin mayor importancia, pero si pierde los datos de todos sus clientes entonces sí estaríamos hablando de un gran problema.

El comportamiento deseado es poder detener el contenedor sin perder los datos almacenados, pero para lograrlo es necesario conocer lo que Docker ha llamado *Volúmenes*. Continue leyendo la próxima sección para conocer cómo resolver el problema.

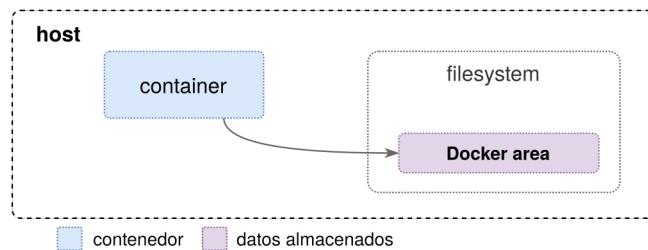
## Persistencia de datos en volúmenes

Los volúmenes son el mecanismo diseñado para persistir los datos utilizados o generados por los contenedores. Cada volumen corresponde con un espacio en disco gestionado por Docker (`/var/lib/docker/volumes/` para los sistemas Linux).

Los volúmenes están fuera del ciclo de vida de los contenedores, lo que significa que puede detener y eliminar los contenedores sin preocuparse de la pérdida de la información. De hecho, puede crear el volumen antes de iniciar el contenedor, e incorporar los ficheros necesarios.

Los beneficios de contar volúmenes son:

- Poder detener el contenedor sin perder la información.
- Gestionar el espacio en disco directamente desde los comandos de Docker.
- Realizar salvas y restauraciones de la información de forma fácil y rápida.
- Compartir un mismo volumen con múltiples contenedores.



Representación de los volúmenes en Docker

Entre los múltiples comandos en docker, existe uno para gestionar los volúmenes. Utilice un par de minutos para conocerlo y saber su descripción.

```
$ docker volume --help
```

## Crear volumen

El próximo paso va a ser crear un volumen para utilizarlo en el ejemplo de Redis. El objetivo es preparar el escenario para no perder los datos de las llaves almacenadas.

```
$ docker volume create redis-data
```

```
redis-data
```

Liste los volúmenes y encuentre el creado por usted hace unos segundos.

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	redis-data

Utilice el comando `inspect` para conocer los detalles del volumen `redis-data`. El resultado muestra la ubicación en disco donde son almacenados los datos, este espacio es gestionado directamente por Docker, por lo que otros sistemas no pueden acceder al directorio.

```
$ docker volume inspect redis-data
```

```
[  
  {  
    "CreatedAt": "2022-04-03T18:26:30Z",  
    "Driver": "local",  
    "Labels": {},  
    "Mountpoint": "/var/lib/docker/volumes/redis-data/_data",  
    "Name": "redis-data",  
    "Options": {},  
    "Scope": "local"  
  }  
]
```

## Montar el volumen al iniciar del contenedor

El volumen `redis-data` está creado, ahora debe ser vinculado al contenedor de Redis durante el arranque. Existen dos parámetros habilitados para realizar el vínculo: `--mount` y `--volume`. Ambas formas son válidas, pero en el ejemplo se utiliza `--mount` por ser más descriptivo.

```
$ docker container run \
--detach \
--network redis \
--mount type=volume,src=redis-data,dst=/data \
redis:6.2.6
```

```
61c8d781bc51337419374bb6704520aad703c108020525573ef9947896eb9dd4
```

El nuevo comando es bastante intuitivo, pero de todas formas analicemos su contenido. El valor del comando `--mount` es la unión de múltiples elementos llave/valor. *Type* define la manera de montar los ficheros, existen tres posibles valores: *volume*, *bind* y *tmpfs*.

Los valores *src* y *dst* definen origen y destino de la información. En este caso, el origen es el nombre del volumen, y el destino es la carpeta `/data` dentro del contenedor. El resultado es el montaje del directorio origen en la dirección de destino. Con la unión de ambos puntos se garantiza la persistencia de los datos en el volumen `redis-data`.

Utilice nuevamente el comando `inspect` sobre el contenedor iniciado, analice la estructura de montaje y compruebe la relación existente entre ambos puntos, *src* y *dst*.

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container inspect $CONTAINER_ID --format '{{ json .Mounts }}' | jq

[
{
  "Type": "volume",
  "Name": "redis-data",
  "Source": "/var/lib/docker/volumes/redis-data/_data",
  "Destination": "/data",
  "Driver": "local",
  "Mode": "z",
  "RW": true,
  "Propagation": ""
}
]
```

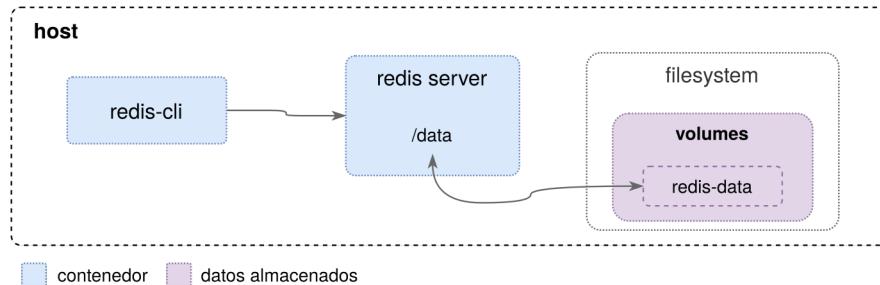
## Comprobar la persistencia

Los próximos comandos adicionan la llave en el servidor.

```
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID
```

```
$ 61c8d781bc51:6379> set mykey somevalue
OK
```

```
$ 61c8d781bc51:6379> get mykey
"somevalue"
```



Ejemplo del servidor Redis utilizando volúmenes

Salga del contenedor utilizando `Ctrl+C` y detenga el servidor de Redis.

```
$ docker container stop $CONTAINER_ID
```

```
61c8d781bc51
```

Inicie un nuevo servidor montando el mismo volumen. Una vez dentro consulte el valor de la llave.

```
$ docker container run \
--detach \
--network redis \
--mount type=volume,src=redis-data,dst=/data \
redis:6.2.6
```

```
0f16a8653f7201a9e65e7c2b457e97700dfbc7afcd0412eb47881df65031aff
```

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID
```

```
$ 0f16a8653f72:6379> get mykey
"somevalue"
```

¡Enhorabuena! Si ha logrado obtener la respuesta "somevalue" significa que ha logrado persistir la información del servidor Redis. A partir de este momento no tiene que preocuparse por la pérdida de datos en sus contenedores.

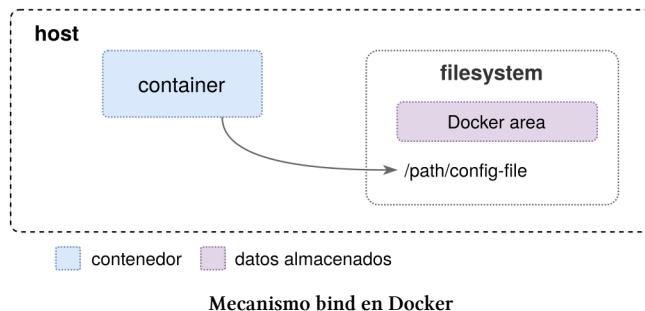
## Diferentes configuraciones por entorno

Los volúmenes son el mecanismo preferido para almacenar los datos generados en un contenedor, pero hay momentos donde es mejor cambiar la estrategia.

Cuando los sistemas necesitan definir muchas configuraciones, utilizan ficheros para agruparlas todas en un mismo sitio. Seguramente ha accedido alguna vez a un servidor de bases de datos para modificar un parámetro de memoria, el número de clientes concurrentes o algún otro detalle. Estas configuraciones son, en muchos casos, las que marcan la diferencia entre los entornos de desarrollo<sup>63</sup>.

Utilizar volúmenes para almacenar ficheros de configuración no es buena idea. Va a necesitar editar el fichero dentro del contenedor, lo cual es muy engorroso porque la imagen no tiene los editores adecuadas. Para solucionar el problema necesita un mecanismo distinto de montar ficheros llamado *bind* (atadura o enlace).

El mecanismo *bind* permite incluir ficheros y carpetas desde la máquina hacia el contenedor. Para hacer referencia a los ficheros en la máquina es necesario utilizar el camino absoluto de cada uno. La ventaja de este mecanismo es poder utilizar el editor de texto de su preferencia para modificar, de forma rápida y sencilla, los ficheros de configuración.



Veamos un ejemplo práctico de cómo utilizar el mecanismo *bind* con el servicio de Redis.

Redis<sup>64</sup> almacena sus configuraciones en el fichero `redis.conf`, sin embargo este fichero no existe cuando se inicia el contenedor. Compruebe esta afirmación iniciando el servicio de Redis y consultando los registros impresos durante el arranque.

<sup>63</sup>Los entornos de desarrollo pueden ser desarrollo, pruebas y producción.

<sup>64</sup>[https://hub.docker.com/\\_/redis](https://hub.docker.com/_/redis)

```
$ docker container run \
--detach \
--network redis \
redis:6.2.6

93099f32f92b9d161e05699953bc8a8b15e072613bd09a4dacaee9c174706bf6

$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container logs $CONTAINER_ID

1:C 05 Apr 2022 21:49:25.968 # o000o000o000o Redis is starting o000o000o000o
1:C 05 Apr 2022 21:49:25.968 # Redis version=6.2.6, bits=64, commit=00000000, modified=0, \
pid=1, just started
1:C 05 Apr 2022 21:49:25.968 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
```

En las últimas líneas se menciona la no existencia del fichero de configuración porque se están utilizando los valores por defecto. Acceda al servidor de Redis desde un contenedor en modo cliente para conocer el valor del parámetro `maxmemory`.

```
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID

$ 93099f32f92b:6379> info memory
...
maxmemory:0
...
```

El valor que aparece es cero, lo que significa que no hay un límite máximo de memoria para el servidor. Teniendo esto en cuenta, el objetivo del ejercicio va a ser crear un fichero `redis.conf` con el valor máximo de memoria en 30 megas. Acto seguido el fichero va a ser montado en el contenedor utilizando el mecanismo *bind*.

Detenga el contenedor creado anteriormente.

```
$ docker container stop $CONTAINER_ID
```

Cree el fichero `redis.conf` con la nueva configuración de memoria máxima.

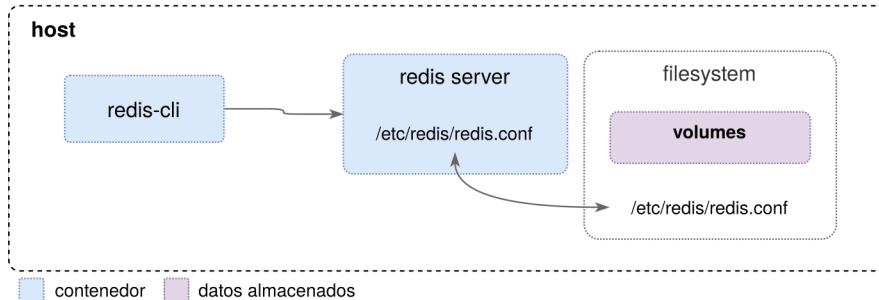
```
$ cat <<EOF >>redis.conf
maxmemory 30mb
EOF
```

Inicie el servidor Redis montando el fichero `redis.conf`. Este comando incluye el parámetro `--mount` especificando el tipo de montaje *bind*. También se ha cambiado el comando *Cmd* para indicarle a Redis que utilice el fichero de configuraciones enlazado a la máquina.

```
$ docker container run \
--detach \
--network redis \
--mount type=bind,src="$(pwd)"/redis.conf,dst=/etc/redis/redis.conf \
redis:6.2.6 redis-server /etc/redis/redis.conf
```

4098cddc3ff10452fef131ab12c5f2cc1e7d6338a369c4230d952e8ef2120f11

```
$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container logs $CONTAINER_ID
...
Configuration loaded
...
```



Ejemplo del mecanismo bind con Redis

Este último mensaje indica que el fichero ha sido cargado correctamente, pero nunca está demás hacer un doble chequeo. Vuelva a acceder con el cliente para comprobar el máximo valor de memoria permitido.

```
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID

$ 93099f32f92b:6379> info memory
...
maxmemory:31457280
...
```

Perfecto, en este momento el valor de memoria máxima ha cambiado, lo que significa que Redis está utilizando el fichero de configuración creado. Ahora puede cambiar los valores a su antojo de forma fácil, también puede ajustar las diferencias entre sus entornos de desarrollos sin cambiar la imagen.

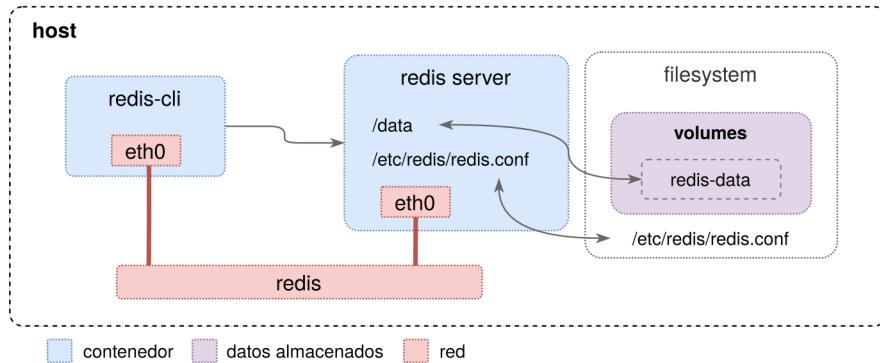
## Configuraciones y volúmenes en el mismo contenedor

Nada le impide utilizar múltiples volúmenes en un mismo contenedor. De hecho, puede realizar todas las combinaciones de volúmenes y configuraciones que desee al mismo tiempo.

Cuando lleve los conocimientos a la práctica, va a ver que no hacen falta tantos volúmenes para un contenedor. La filosofía asociada a contenedores recomienda tener una responsabilidad por contenedor, o lo que es lo mismo, una sola aplicación. Las aplicaciones pueden necesitar un volumen para persistir sus datos, pero en pocas ocasiones necesitan dos.

Por otro lado, las configuraciones se comportan de forma distinta, porque es frecuente tener múltiples ficheros para configurar un servicio. La primera solución puede ser un punto de montaje para cada fichero, pero si todos están en la misma carpeta, puede montar directamente la carpeta con todos los ficheros dentro. De esta forma reduce los parámetros de configuración.

El sistema Redis es un buen ejemplo para poner en práctica los dos tipos de volúmenes vistos. Utilice los comandos que aparecen a continuación para llevarlo a cabo.



Configuraciones, volúmenes y redes

```
$ docker container run \
--detach \
--network redis \
--mount type=volume,src=redis-data,dst=/data \
--mount type=bind,src="$(pwd)"/redis.conf,dst=/etc/redis/redis.conf \
redis:6.2.6 redis-server /etc/redis/redis.conf

a8d9199e972d621106bbea8ebbf78d357139b58afac10d924fcc48dccf8586

$ CONTAINER_ID=$(docker container ls --latest --quiet)
$ docker container run \
--network redis \
-it \
redis:6.2.6 redis-cli -h $CONTAINER_ID

$ a8d9199e972d:6379> info memory
maxmemory:31457280

$ a8d9199e972d:6379> get mykey
"somevalue"
```

Note un detalle interesante, al utilizar el volumen `redis-data` ha incorporado la información almacenada en ejercicios anteriores, de ahí que se muestre el valor de la llave `mykey`.

## Eliminar volúmenes y recuperar espacio

Crear espacios para persistir la información es importante, pero igual de importante es saber cómo recuperar el espacio utilizado. Recuerde, el espacio en disco es un recurso finito, hay que optimizarlo siempre que sea posible.

Los volúmenes no forman parte del ciclo de vida de los contenedores, por lo tanto, al detener o eliminar un contenedor, el volumen sigue existiendo y ocupando espacio. Por tal motivo, es necesario hacer uso de nuevos comandos a la hora de eliminar los volúmenes.

El comando `system df` le va a mostrar cuánto espacio puede eliminar y a cuál concepto pertenece. En este momento, como estamos hablando de volúmenes, la fila de interés es *Local Volumes*.

```
$ docker system df
```

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	51	5	15.68GB	15.36GB (97%)
Containers	86	1	31.61MB	31.61MB (100%)
Local Volumes	1	1	780B	0B (0%)
Build Cache	104	0	468B	468B

El valor más significativo de todas las columnas es el espacio a reclamar (*RECLAIMABLE*). En esta ocasión no hay un valor para los volúmenes porque apenas se ha escrito una llave en Redis, pero si realiza muchas operaciones va notar la diferencia.

Suponga que tiene un gran volumen disponible para ser eliminado, en ese caso, utilice el comando `rm` para eliminar los datos.

```
$ docker volume rm redis-data
```

```
Error response from daemon: remove redis-data: volume is in use - [0f16a8653f7201a9e65e7c\2b457e97700dfbc7afcd0412eb47881df65031aff, 2667c41417310755d27c4c698395732ae2d1089a4acda\108a3f2ff2344ac886f, 61c8d781bc51337419374bb6704520aad703c108020525573ef9947896eb9dd4, a8\ d9199e972d621106bbea8ebbf78d357139b58aface10d924fcc48dccf8586]
```

Si recibe un mensaje de error similar al ejemplo, significa que el volumen está siendo utilizado por un contenedor, por lo tanto no puede eliminarse. Va a necesitar detener y eliminar el contenedor para luego borrar sus datos. Este mecanismo de protección es de mucha ayuda para evitar errores humanos.

Liste los contenedores en funcionamiento, busque el identificador de Redis y detenga el servicio. Si lo desea, puede detener todos los contenedores en ejecución al mismo tiempo con un solo comando.

```
$ docker container stop $(docker container ls --quiet)
```

```
a8d9199e972d
```

En realidad son dos comandos de Docker en una sola línea. El comando entre paréntesis obtiene el listado de todos los *IDs*. Estos *IDs* se pasan como parámetros de entrada al comando `stop`. En la salida del comando `stop` se muestran los *IDs* de los contenedores que fueron detenidos.

Elimine todos los contenedores que estén detenidos con el comando `prune`.

```
$ docker container prune
```

```
WARNING! This will remove all stopped containers.  
Are you sure you want to continue? [y/N] y
```

Si en vez de eliminar todos los contenedores, desea eliminar directamente el contenedor de Redis, utilice el comando `rm` seguido del identificador del contenedor.

```
$ docker container rm CONTAINER-ID
```

Después de haber eliminado el contenedor está en condiciones de eliminar el volumen sin que se muestren errores.

```
$ docker volume rm redis-data
```

```
redis-data
```

Utilice el comando `prune` en caso de necesitar borrar todos los volúmenes a la vez.

```
$ docker volume prune --force
```

Perfecto, ha logrado eliminar toda la información. Este paso no puede ser revertido, por lo tanto, tenga mucho cuidado cuando lo ejecute en entornos de producción.

## Ejercicios para practicar lo aprendido

Después de haber incorporado nuevos conceptos y comandos es momento de practicar. Los ejercicios que aparecen a continuación tienen la intención pasar un rato agradable delante del Terminal.

1. Crear la red `mysql` con el controlador `bridge`.
2. Crear el volumen `mysql`.
3. Inicie un servidor con la imagen [mariadb:10.7.3<sup>65</sup>](https://hub.docker.com/_/mariadb). Utilice las variables de entorno `MARIADB_USER=erasedunavez`, `MARIADB_PASSWORD=docker` y `MARIADB_ROOT_PASSWORD=invisible`. Monte el volumen creado anteriormente en la carpeta `/var/lib/mysql`. Inicie el servicio dentro de la red creada.
4. Consulte los *logs* del servidor iniciado en el paso 1 para comprobar que todo ha ido bien.
5. Inicie un contenedor con la misma imagen de MariaDB en modo cliente. Luego realice una conexión hacia el servidor iniciado en el ejercicio anterior. Tenga en cuenta que tendrá que hacer uso del usuario y contraseña definidos en las variables de entorno.
6. Desde el contenedor cliente adicione un usuario con el nombre `docker`.
7. Detenga y elimine el contenedor del servidor. Luego inicie uno nuevo con las mismas configuraciones para comprobar que el usuario `docker` existe en MariaDB.

---

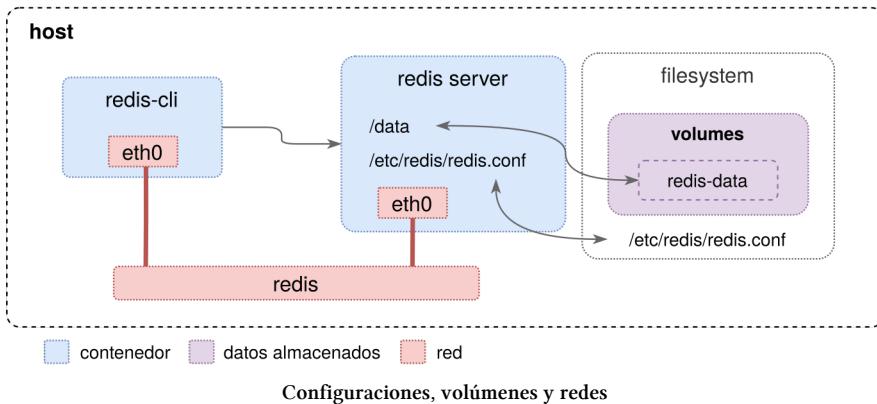
<sup>65</sup>[https://hub.docker.com/\\_/mariadb](https://hub.docker.com/_/mariadb)

## Resumen del capítulo

Durante el capítulo se explicaron los principales conceptos asociados a las configuraciones y a la persistencia de la información. Ambos elementos están estrechamente relacionados con los mecanismos brindados por Docker para montar ficheros, carpetas o volúmenes.

Se mostró a través de ejemplos la importancia de persistir la información utilizando volúmenes. El mecanismo de persistencia es posible gracias a que los volúmenes existen fuera del ciclo de vida de los contenedores.

Las configuraciones a través de ficheros o carpetas no reemplazan a los volúmenes, todo lo contrario, se complementan para brindar gran flexibilidad a los contenedores a través de diferentes entornos.



Los ejercicios realizados en la sección anterior buscan practicar los conocimientos aprendidos, pero es necesario más tiempo de entrenamiento para dominar la persistencia en contenedores. Le recomiendo explorar otros servicios que necesiten almacenar información para desarrollar esta habilidad. Como sugerencia puede utilizar [PostgreSQL<sup>66</sup>](https://hub.docker.com/_/postgres), [MongoDB<sup>67</sup>](https://hub.docker.com/_/mongo) y [RabbitMQ<sup>68</sup>](https://hub.docker.com/_/rabbitmq).

<sup>66</sup>[https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres)

<sup>67</sup>[https://hub.docker.com/\\_/mongo](https://hub.docker.com/_/mongo)

<sup>68</sup>[https://hub.docker.com/\\_/rabbitmq](https://hub.docker.com/_/rabbitmq)

# Construcción de imágenes

Las imágenes son el punto de inicio de los contenedores, son la base de ficheros utilizados por los contenedores en su ejecución. Si la base es buena, los procesos van a funcionar correctamente, pero si tiene fisuras, el servicio va dar muchos dolores de cabeza.

Crear correctamente las imágenes de contenedores es un reto. Son muchas las tecnologías utilizadas a día de hoy y cada una con sus características. Todas las aplicaciones están preparadas para ser empaquetadas en imágenes, pero no vale solamente hacer uso de ellas, también es importante crear nuevas imágenes.

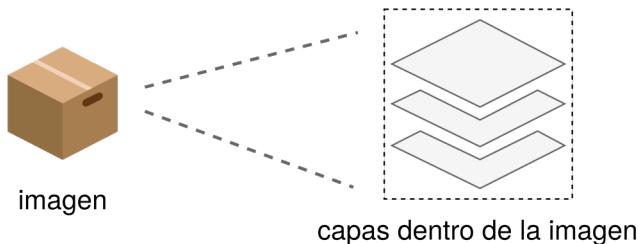
A la hora de crear imágenes, existen buenas prácticas y principios generales aplicables a todas las tecnologías. Durante el capítulo va a aprender estos aspectos aplicables a todas las imágenes. Para avanzar con paso firme va a ser necesario recordar [el concepto de imagen](#) explicado en secciones anteriores.

Hasta el momento se han utilizado las imágenes para iniciar contenedores, siempre se han visto como estructuras cerradas listas para ejecutarse, pero este capítulo es diferente. *¡Llegó el momento de construir sus propias imágenes!*

## Las imágenes se estructuran por capas

Hemos dicho en múltiples ocasiones que las imágenes son un conjunto de ficheros, lo cual es cierto, pero también es necesario saber que estos ficheros están agrupados o estructurados en capas (*layers*).

Las capas están compuestas por múltiples ficheros. Los ficheros son generados a partir de instrucciones ejecutadas durante el proceso de construcción de la imagen. Docker le asigna un identificador a cada capa para diferenciarlas, este identificador está codificado con el algoritmo *SHA256*. Por último, la superposición de capas da lugar a las imágenes de los contenedores.



Estructura de capas de la imagen

Los conceptos de capas y grupos de ficheros pueden parecer abstractos y complejos. No se preocupe si le cuesta trabajo entenderlo a la primera, poco a poco se va a familiarizar con estos temas al avanzar en las secciones. En las rutinas diarias con contenedores no se utiliza mucho el concepto de *capas*, pero tener este conocimiento le va a ayudar a construir imágenes de forma eficiente.

En esta sección se van a mencionar aspectos básicos de las capas en los contenedores, pero si desea ampliar la información, puede utilizar [documentación oficial](#)<sup>69</sup>. Pasemos a tocar con las manos el concepto de *Capas* a través de los siguientes comandos.

Elimine la imagen *erase-una-vez-1* de su máquina para borrar todas las capas asociadas a ella.

```
$ docker image rm ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
```

Descargue nuevamente la imagen como si fuera la primera vez. Fíjese en los comentarios incluidos en la salida, la imagen está compuesta por tres capas, una de ellas existía en el sistema y las otras dos tuvieron que ser descargadas. Esta es una de las ventajas de la estructuración en capas, solamente va a descargar los bloques que necesite y nada más. Este funcionamiento optimiza el espacio en disco, así como también acelera los procesos de descarga y construcción de imágenes.

```
$ docker image pull ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
```

```
v0.3.0: Pulling from mmorejon/erase-una-vez-1
069a56d6d07f: Already exists          # <- Capa de la imagen
6a02908ff039: Pull complete          # <- Capa de la imagen
26019ab338e4: Pull complete          # <- Capa de la imagen
Digest: sha256:8d1abf6bdc634d4ac78422d376fd55962b2bfe3c81a8c45d476b598c11c930b6
Status: Downloaded newer image for ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
```

Utilice el comando *save* para exportar la información de la imagen a un archivo *tar*.

```
$ docker image save \
-o erase-una-vez-1.tar ghcr.io/mmorejon/erase-una-vez-1:v0.3.0
```

Liste los elementos de su directorio para comprobar que existe el fichero *erase-una-vez-1.tar*. Luego cree la carpeta *erase-una-vez-1-image* y extraiga la información de la imagen en ella.

```
$ mkdir erase-una-vez-1-image
$ tar xvf erase-una-vez-1.tar -C erase-una-vez-1-image
```

Acceda a la carpeta *erase-una-vez-1-image* y compruebe que existen tres carpetas. Cada carpeta corresponde con una capa, y entre todas ellas componen la imagen. Dentro de cada carpeta hay un fichero *layer.tar* con los ficheros de cada capa.

---

<sup>69</sup><https://docs.docker.com/storage/storagedriver/>

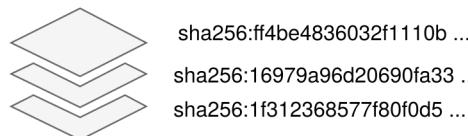
```
$ tree erase-una-vez-1-image
```

```
erase-una-vez-1-image
├── 0ea9f89692c83558320491d8c6799d1305f6ae336c72137f6c3620ef9b13a97f
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── 2d575f0b8481e4c476ada9bab76551e8e0c9439089ed94d0ed5f51d52cd2ace2
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── 4f1101dd1eb24fe4a4712e939e95c86380a15f1ddb64136a64b88da53e438d57.json
└── 7aa62770f20b354a41fc388ce46740e49556c8c5225892c663a2143d767582bf
    ├── VERSION
    ├── json
    └── layer.tar
└── manifest.json
└── repositories
```

Utilice el comando `inspect` para consultar el número de capas que componen la imagen. El resultado son los identificadores de las tres capas codificados con el algoritmo *sha256*.

```
$ docker image inspect ghcr.io/mmorejon/erase-una-vez-1:v0.3.0 \
--format '{{ .RootFS.Layers }}' | jq

[
  "sha256:ff4be4836032f1110bf03d4a17eeba4d126a1910cf0377386f2bd288469e7cfe",
  "sha256:16979a96d20690fa33cde3ea2f125b262c8ac4ce2b60c78f70741ec0d79f6d35",
  "sha256:1f312368577f80f0d518240893d013f769782728fd6cea747cc52b9145f23742"
]
```



ghcr.io/mmorejon/erase-una-vez-1:v0.3.0

Capas de la imagen erase-una-vez-1

El siguiente comando le va a permitir generar el identificador de una capa de forma manual, para luego comprobar la correspondencia del código con los ficheros *layers.tar* de las carpetas.

```
$ shasum -a 256 erase-una-vez-1-image/7fda5d5233a852177110f9f2b4468f88ffd368230bb137d1f1f\ff1263d4e51b6/layer.tar
```

```
43c0bb0e95acbd95afb0d97c2fe1ee658fef3ce7fec2666aad6933ef1cf72e20  erase-una-vez-1-image/7\fda5d5233a852177110f9f2b4468f88ffd368230bb137d1f1ff1263d4e51b6/layer.tar
```

Después de terminar las comprobaciones puede borrar la carpeta *erase-una-vez-1-image* porque no la volveremos a necesitar. La intención era mostrarles que la imagen es algo tangible, porque son ficheros.

## Relación entre Instrucciones y Capas

Los ficheros que componen cada capa son generados a partir de instrucciones. Las instrucciones son palabras reservadas utilizadas durante el proceso de construcción de la imagen, p.ej: *FROM*, *COPY*, *RUN*, *ENTRYPOINT*. Cada palabra reservada implica una acción, sin embargo, no todas generan ficheros, pero para aquellas que si lo hacen, Docker agrega una capa por instrucción.

El comando *history* muestra todas las instrucciones utilizadas para crear la imagen. En los casos donde no se generan ficheros el tamaño es cero, y por lo tanto no se generan capas. El total de filas con valores mayores a cero va a corresponder con el número de capas, en este caso tres.

```
$ docker image history ghcr.io/mmorejon/erase-una-vez-1:v0.3.0 \
--format 'table {{.ID}}\t{{.CreatedBy}}\t{{.Size}}'
```

IMAGE	CREATED BY	SIZE
127c6d150a71	ENTRYPOINT ["erase-una-vez-1"]	0B
<missing>	USER elf	0B
<missing>	COPY /go/bin/erase-una-vez-1 /usr/local/bin/...	1.44MB
<missing>	COPY /etc/passwd /etc/passwd # buildkit	1.21kB
<missing>	LABEL org.opencontainers.image.source=https:...	0B
<missing>	LABEL language=golang	0B
<missing>	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B
<missing>	/bin/sh -c #(nop) ADD file:80bf8bd014071345b...	5.61MB

Después de conocer la estructura en capa de las imágenes el siguiente paso son las instrucciones. Las próximas secciones explican las instrucciones más utilizadas en Docker, pero también van a permitirle responder la siguiente pregunta:

*¿Dónde se guardan las instrucciones que generan la imagen?*

## La importancia del fichero Dockerfile

La industria del software ha demostrado en múltiples ocasiones la importancia de registrar las acciones de un procedimiento en ficheros. Entre las principales ventajas se están:

- Mayor claridad / visibilidad de las acciones.
- Permite incluirlo en el sistema de control de versiones.
- Permite la automatización del proceso.

Docker no podía pasar por alto estas buenas prácticas y decidió utilizar un fichero como mapa para crear las imágenes de los contenedores. A este fichero se le dio el nombre *Dockerfile*, y su objetivo principal es almacenar las instrucciones que genera una imagen.

En la actualidad es muy común ver este fichero en la raíz de los proyectos como parte del código fuente. Su presencia indica la posibilidad de crear una imagen del código para luego desplegarlo como contenedor. Al abrir el fichero va a identificar, de forma rápida, todas las dependencias y configuraciones necesarias para ejecutar el proyecto. Aquí le comparto un [enlace de ejemplo](#)<sup>70</sup>.

Tener el fichero *Dockerfile* dentro del repositorio de código trae muchas ventajas, pero tenga cuidado de **no incluir** información sensible como pueden ser: credenciales, tokens o algún otro valor similar utilizado para instalar paquetes o dependencias.

Alrededor del fichero *Dockerfile* existen buenas prácticas que le van a permitir sacar provecho de este fichero. Muchas de estas buenas prácticas son comentadas durante el transcurso del capítulo. Adicional a los conceptos explicados en el libro está la [documentación oficial de Docker](#)<sup>71</sup>, que siempre vale la pena consultarla.

Ahora que conoce el propósito del fichero *Dockerfile*, nada le impide crear su primera imagen.

## Crear una imagen desde cero

La primera imagen a crear es sencilla e interesante, porque se va a construir desde cero. La filosofía para crear esta primera imagen aparece a continuación.

El punto de partida siempre es otra imagen existente, y se utiliza la instrucción *FROM* para definir esta imagen base. Todos los ficheros *Dockerfile* cuentan con al menos una instrucción *FROM*, y todas las imágenes pueden ser utilizadas como base para construir otras nuevas. En la primera imagen se va a construir una imagen desde cero, por lo tanto se va a utilizar la imagen *scratch*<sup>72</sup>.

La imagen *scratch* es especial porque no contiene ficheros, está completamente vacía. Puede [consultar este enlace](#)<sup>73</sup> para conocer los detalles que hacen especial esta imagen. Después de definir el punto de partida, lo siguiente es adicionar la aplicación que va a funcionar dentro del contenedor. Para este ejemplo va a adicionar el binario *erase-una-vez-1*<sup>74</sup>, este va a ser el único fichero dentro de la imagen. Por último se definen los comandos utilizados por el contenedor para iniciar la aplicación.

Los elementos mencionados en flujo de creación se encuentran en GitHub. Descargue el repositorio de actividades del libro y acceda a la carpeta *scratch* para hacer uso de ellos.

<sup>70</sup><https://github.com/mmorejon/erase-una-vez-1>

<sup>71</sup><https://docs.docker.com/engine/reference/builder/>

<sup>72</sup>[https://hub.docker.com/\\_/scratch](https://hub.docker.com/_/scratch)

<sup>73</sup><https://github.com/mmorejon/knowing-scratch-image>

<sup>74</sup><https://github.com/mmorejon/erase-una-vez-1>

```
$ git clone git@github.com:mmorejon/erase-una-vez-docker.git
$ cd erase-una-vez-docker/scratch
```

Dentro está el fichero *Dockerfile* y el binario de la aplicación *erase-una-vez-1*. Imprima el fichero *Dockerfile* en pantalla para analizar su contenido.

```
$ cat Dockerfile

FROM scratch
COPY erase-una-vez-1 /
ENTRYPOINT [ "./erase-una-vez-1"]
```

La instrucción *FROM* establece el punto de inicio de la imagen. Luego aparece el comando *COPY*, responsable de copiar el binario desde el directorio local hacia el directorio raíz de la imagen. Por último la instrucción *ENTRYPOINT*, utilizada para definir el comando de inicio del contenedor. Seguramente recuerde el concepto *Entrypoint* cuando se analizó la sección *modificar el comando de inicio*, pues es exactamente el mismo, pero esta vez lo está definiendo desde el fichero *Dockerfile*.

Cada vez que utilice la instrucción *COPY* se crea una capa en la imagen. En el ejemplo se utiliza una vez, por lo tanto la imagen tiene una sola capa. Dentro de esta capa va a estar el binario de la aplicación *erase-una-vez-1*<sup>75</sup>.

Tras declarar todos los pasos en el *Dockerfile*, es necesario transformar esas declaraciones en una imagen. Para lograrlo se utiliza el comando *build*, que significa construir la imagen.

```
$ docker image build --tag docker-book-scratch .

[+] Building 0.1s (5/5) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 36B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 38B
=> [1/1] COPY erase-una-vez-1 /
=> exporting to image
=> => exporting layers
=> => writing image sha256:bdeff9d205eae3999d446eab14992c208acef97c9e43b1eba5d35064fd0b8\
a3c
=> => naming to docker.io/library/docker-book-scratch
```

El parámetro *tag* se ha utilizado para establecer el nombre de la imagen, mientras que el punto al final indica el directorio donde se encuentra el fichero *Dockerfile*. Dedique unos minutos para analizar los

---

<sup>75</sup><https://github.com/mmmorejon/erase-una-vez-1>

mensajes mostrados durante la construcción, fíjese en el momento donde se copia el binario, donde se crean las capas y cuando se establece el nombre de la imagen.

Si ejecuta el comando nuevamente va a ver la misma secuencia de pasos. Sin importar el número de veces, la salida va a ser siempre la misma. Esta es una de las ventajas de utilizar el fichero *Dockerfile* en la construcción de imágenes.

Ahora liste las imágenes para conocer el resultado obtenido.

```
$ docker image ls docker-book-scratch
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-book-scratch	latest	bdefff9d205ea	13 minutes ago	1.44MB

Si consulta el tamaño del binario va a notar que coincide con el tamaño de la imagen. Esto no es coincidencia, se debe a que es el único fichero dentro de la imagen. También puede utilizar el comando *history* para ver los pasos utilizados en la creación de la imagen, estos pasos van a coincidir con las instrucciones del fichero *Dockerfile*.

```
$ docker image history docker-book-scratch \
--format 'table {{.ID}}\t{{.CreatedBy}}\t{{.Size}}'

IMAGE          CREATED BY          SIZE
bdefff9d205ea  ENTRYPOINT [ "./erase-una-vez-1" ]  0B
<missing>      COPY erase-una-vez-1 / # buildkit  1.44MB
```

En el resultado se puede apreciar las dos instrucciones, *COPY* y *ENTRYPOINT*. De estas dos instrucciones solamente *COPY* genera una nueva capa con el tamaño del binario. Como último paso se va a iniciar un contenedor a partir de la imagen creada, esta es la prueba de fuego para ver si su imagen funciona correctamente.

```
$ docker container run --rm docker-book-scratch

hostname: 6fa33293c42e - érase una vez ...
hostname: 6fa33293c42e - érase una vez ...
hostname: 6fa33293c42e - érase una vez ...
```

Si ha obtenido los mismos mensajes en su pantalla significa que todo ha ido bien. En su caso el nombre del *hostname* va a ser diferente porque corresponde con el identificador el contenedor, pero el resto debe ser igual. ¡Enhorabuena! Ha creado su primera imagen de contenedor.

## Comprendiendo el comando **build**

Después de utilizar el comando `build` para construir la primera imagen, se hace necesario conocer algunos detalles sobre su funcionamiento.

La esencia del comando `build` es construir imágenes, pero la forma de construirlas ha cambiado con el tiempo. El primer comando con este propósito fue `docker build`, de hecho todavía puede verlo si consulta la ayuda. Luego, con la llegada de los comando de gestión se asoció esta funcionalidad a las imágenes, por lo tanto el comando pasa a ser `docker image build`.

Sin embargo, la historia no termina aquí, Docker desarrolló un plugin llamado `Buildx`<sup>76</sup> donde aumenta el número de funcionalidades disponibles durante el proceso de construcción. Este plugin es ahora el recomendado para construir imágenes y para suerte de todos, está incluido en el paquete de instalación de Docker.

Antes de utilizarlo, viene bien revisar si existe actualmente en su máquina.

```
$ docker info --format '{{ json .ClientInfo.Plugins }}' | jq  
[  
 {  
   "SchemaVersion": "0.1.0",  
   "Vendor": "Docker Inc.",  
   "Version": "v0.8.1",  
   "ShortDescription": "Docker Buildx",  
   "Name": "buildx",  
   "Path": "/usr/local/lib/docker/cli-plugins/docker-buildx"  
 },  
 ]
```

Si se muestra como instalado significa que puede utilizar el parámetro `--help` para ver todos los subcomandos. Entre las opciones disponibles va estar el comando `build` con las mejoras incluidas.

```
docker buildx --help
```

En este momento usted cuenta con dos formas de construir imágenes en su ordenador, pero solo una es la recomendada. Para evitar confusiones, Docker permite establecer el uso del plugin `buildx` de forma predeterminada. De esta manera va a poder hacer uso de sus bondades a través del comando `image build`.

Si utiliza Docker Desktop el plugin `buildx` está establecido por defecto. Acceda a *Preferences > Docker Engine*, la opción `buildkit` debe tener el valor `true`. En el caso de Docker Engine tiene que crear un fichero llamado `daemon.json`, luego necesita reiniciar el servicio de Docker para utilizar los cambios realizados.

---

<sup>76</sup><https://docs.docker.com/buildx/working-with-buildx/>

```
$ sudo cat <<EOF >/etc/docker/daemon.json
{ "features": { "buildkit": true } }
EOF

$ sudo systemctl restart docker.service
```

En el objetivo de esta sección no es profundizar en el fichero *daemon.json*, pero si le da curiosidad le recomiendo leer [este artículo<sup>77</sup>](#). También puede conocer más sobre cómo activar *BuildKit* en esta otra referencia<sup>78</sup>.

Construya nuevamente la imagen del ejemplo anterior. Si obtiene la misma salida mostrada en el ejemplo significa que está utilizando *BuildKit*. Otra forma de estar seguros es poniendo delante del comando la variable *DOCKER\_BUILDKIT*, si le asigna el valor uno se activa el plugin, y si le asigna el valor cero se desactiva. Ejecute ambos comandos que aparecen a continuación y note las diferencias en las salidas.

```
$ DOCKER_BUILDKIT=1 docker image build --tag docker-book-scratch .

$ DOCKER_BUILDKIT=0 docker image build --tag docker-book-scratch .
```

El uso de *BuildKit* es fundamental para sacar provecho de las futuras funcionalidades en Docker, por lo tanto, le recomiendo mantener activado su uso.

## Conociendo el fichero *.dockerignore*

¿Qué es el fichero *.dockerignore*? y ¿Por qué existe?

Para responder estas preguntas es necesario recordar la arquitectura cliente - servidor de Docker. Cuando Docker inicia la construcción de la imagen, el cliente recolecta todos los ficheros ubicados en la carpeta pasada por parámetros en el comando *build*. En el ejemplo anterior, la ubicación de la carpeta se establece con el punto final. Luego, los ficheros cargados en el cliente se envían al servidor (*docker daemon*) para realizar la construcción.

Pero, ¿qué sucede si en esa carpeta existen ficheros o carpetas de gran tamaño? Estos ficheros van a ser cargados al cliente y luego enviados al servidor. Al ser grandes estos ficheros va a aumentar el consumo de RAM y el tiempo de procesamiento durante la construcción, siendo ambos gastos no deseados. Sin embargo, aunque Docker no necesite estos ficheros, puede que sean necesarios para otro propósito y por eso están en el repositorio. Entonces ¿cómo se puede evitar cargar estos ficheros en el proceso de construcción?

Para solucionar el problema Docker ha diseñado el fichero *.dockerignore* siguiendo la misma filosofía de la industria p.ej *.gitignore*. Este fichero se va a utilizar para excluir los elementos presentes en la

<sup>77</sup><https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file>

<sup>78</sup>[https://docs.docker.com/develop/develop-images/build\\_enhancements/#to-enable-buildkit-builds](https://docs.docker.com/develop/develop-images/build_enhancements/#to-enable-buildkit-builds)

carpeta que no deban formar parte del proceso de construcción. La estructura interna del fichero sigue las mismas convenciones que el fichero `.gitignore`. Si desea profundizar en este tema [consulte el sitio oficial en Docker](#)<sup>79</sup>.

En el repositorio de ejercicios del libro se ha diseñado un caso para mostrar cómo quedan excluidos los ficheros utilizando el fichero `.dockerignore`.

```
$ git clone git@github.com:mmorejon/erase-una-vez-docker.git  
$ cd erase-una-vez-docker
```



En ejercicios previos debe haber utilizado este repositorio, si este es su caso puede saltar el paso de clonar.

Dentro de la carpeta `dockerignore` va a encontrar los ficheros para crear la imagen. El fichero `.dockerignore` ha sido configurado para excluir el fichero `file-one.txt`. El fichero `Dockerfile` creado es muy simple, va a copiar los ficheros del directorio hacia la carpeta `/myfiles`. La imagen base utilizada es [alpine](#)<sup>80</sup>, porque es pequeña y tiene algunos comandos básicos como es `ls`.

```
$ cat ./dockerignore/Dockerfile  
  
FROM alpine:3.15.4  
COPY . /myfiles
```

Utilice el comando `build` para construir la nueva imagen. Tenga en cuenta un detalle, el último parámetro del comando es la ubicación de la carpeta donde están los ficheros dentro del repositorio.

```
$ docker image build --tag docker-book-ignore ./dockerignore  
  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 36B  
=> [internal] load .dockerignore  
=> => transferring context: 53B  
...  
...
```

Luego inicie un contenedor utilizando la imagen construida para listar los ficheros copiados hacia la carpeta `/myfiles`.

<sup>79</sup><https://docs.docker.com/engine/reference/builder/#dockerignore-file>

<sup>80</sup>[https://hub.docker.com/\\_/alpine](https://hub.docker.com/_/alpine)

```
$ docker container run docker-book-ignore ls /myfiles  
  
Dockerfile  
file-two.txt
```

Como era de esperar, el fichero *file-one.txt* no está dentro de la carpeta, ha sido excluido por el fichero *.dockerignore*. Como detalle interesante, el fichero *.dockerignore* tampoco aparece en el listado porque Docker solamente lo utiliza para el proceso de construcción.

Recuerde los conceptos vistos en esta sección cuando tenga muchos ficheros en el directorio. Utilice el fichero *.dockerignore* durante el proceso de construcción para excluir aquellos elementos que no necesite.

## Ejemplo de imagen para un sitio web

La primera imagen fue creada desde cero utilizando la base *scratch*. Esta aproximación es muy útil cuando desea imágenes pequeñas y su aplicación es un binario que no necesita dependencias de otras bibliotecas. Sin embargo, no todos los casos pueden resolverse de esta forma, imagine que quiere desplegar un sitio web, *¿es recomendable utilizar como base scratch?*

La respuesta es no. Para desplegar un sitio web, en la mayoría de los casos, necesita un servidor web para publicar los ficheros que componen el proyecto. El servidor web y los ficheros no pueden comprimirse y ejecutarse desde un mismo fichero como si fuera un binario, por lo tanto habrá que cambiar la estrategia.

Para dar solución al problema se recomienda cambiar la base de la imagen. Para estos casos lo mejor es buscar una imagen base que incluya el funcionamiento de un servidor web, p.ej: [nginx<sup>81</sup>](#), [openresty<sup>82</sup>](#), [httpd<sup>83</sup>](#) u otro de su preferencia. Por último, hay que adicionar los ficheros del sitio web en la carpeta definida por el servidor para su publicación.

En el repositorio de ejercicios se ha preparado la carpeta *website* para mostrar un ejemplo similar al explicado. Si tiene el repositorio clonado no hace falta volverlo a hacer. Realice los comandos que aparecen a continuación desde la raíz del repositorio.

Consulte los ficheros de la carpeta *website*.

---

<sup>81</sup>[https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx)

<sup>82</sup><https://hub.docker.com/r/openresty/openresty>

<sup>83</sup>[https://hub.docker.com/\\_/httpd](https://hub.docker.com/_/httpd)

```
$ tree website  
  
website  
├── Dockerfile  
└── html  
    └── index.html  
  
1 directory, 2 files
```

La carpeta *html* contiene el fichero del sitio que desea publicar, mientras que el fichero *Dockerfile* contiene los pasos para construir la imagen.

```
$ cat website/Dockerfile  
  
FROM nginx:1.21.6  
COPY html /usr/share/nginx/html
```

En esta ocasión se ha utilizado la imagen de [nginx:1.21.6<sup>84</sup>](#) como base para el servidor web. Luego se copia la carpeta *html* hacia la ubicación donde *Nginx* publica los ficheros. En total son dos líneas de código, pero van a ser suficientes porque el resto de configuraciones están incluidas en la imagen base.

Construya la imagen para el sitio web e inicie un contenedor a partir de ella.

```
$ docker image build --tag docker-book-website ./website  
  
$ docker container run --publish 8000:80 docker-book-website
```

Después de iniciado el contenedor acceda a <http://localhost:8000> para consultar el resultado. El comando *run* se ha dejado enlazado a la consola de manera intencionada para ver de forma rápida los *logs*. Si desea iniciar el contenedor como proceso independiente incluya el parámetro *--detach*.

El punto de inicio de la imagen no ha sido *scratch*, sino Nginx. Esto significa que su imagen resultante es la unión de las capas de Nginx, más las agregadas por su fichero *Dockerfile*. Utilice el comando *history* para confirmar este concepto de capas, fíjese que la capa superior corresponde con la instrucción *COPY* de la carpeta *html*.

---

<sup>84</sup>[https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx)

```
$ docker image history docker-book-website \
--format 'table {{.ID}}\t{{.CreatedBy}}\t{{.Size}}'

IMAGE          CREATED BY                                SIZE
2e53c4050729  COPY html /usr/share/nginx/html # buildkit 551B
<missing>      /bin/sh -c #(nop) CMD ["nginx" "-g" "daemon... 0B
<missing>      /bin/sh -c #(nop) STOPSIGNAL SIGQUIT        0B
<missing>      /bin/sh -c #(nop) EXPOSE 80                 0B
<missing>      /bin/sh -c #(nop) ENTRYPOINT ["/docker-entr... 0B
<missing>      /bin/sh -c #(nop) COPY file:09a214a3e07c919a... 4.61kB
<missing>      /bin/sh -c #(nop) COPY file:0fd5fca330dcd6a7... 1.04kB
<missing>      /bin/sh -c #(nop) COPY file:0b866ff3fc1ef5b0... 1.96kB
<missing>      /bin/sh -c #(nop) COPY file:65504f71f5855ca0... 1.2kB
<missing>      /bin/sh -c set -x    && addgroup --system ... 60.1MB
<missing>      /bin/sh -c #(nop) ENV PKG_RELEASE=1~bullseye 0B
<missing>      /bin/sh -c #(nop) ENV NJS_VERSION=0.7.2       0B
<missing>      /bin/sh -c #(nop) ENV NGINX_VERSION=1.21.6     0B
<missing>      /bin/sh -c #(nop) LABEL maintainer=NGINX Do... 0B
<missing>      /bin/sh -c #(nop) CMD ["bash"]                0B
<missing>      /bin/sh -c #(nop) ADD file:9816c9c29627693c3... 74.3MB
```

El sitio web mostrado es un ejemplo sencillo, pero válido. El objetivo ha sido mostrar el uso de imágenes existentes en favor reutilizar el trabajo de la comunidad. Además, recuerde que las imágenes oficiales cuentan con la garantía de calidad y seguridad explicada en secciones anteriores. Por último, siempre que necesite crear una nueva imagen, analice cuál debe ser su punto de inicio.

## Ejemplo de imagen para aplicación con dependencias

Es común utilizar dependencias durante el proceso de desarrollo. Estas dependencias le permiten hacer uso de funcionalidades desarrolladas por terceros ahorrando tiempo de trabajo. De esta forma evita reinventar la rueda todo el tiempo, y va a poder concentrarse en la lógica de su negocio. Este modelo de colaboración existe en todas las comunidades de desarrollo y de seguro usted ya lo ha utilizado, o por lo menos lo conoce.

Este tipo de aplicaciones con dependencias también se despliegan en forma de contenedores, por lo tanto, va a ser necesario poder empaquetar este tipo de aplicaciones en imágenes. Para poner esto en práctica se ha creado [una aplicación<sup>85</sup>](https://github.com/mmorejon/erase-una-vez-docker/tree/main/app) en la carpeta *app*. Es un sistema simple, pero es válido para el objetivo que se busca en este momento. La única funcionalidad de la aplicación es listar las imágenes de contenedores descargadas en la máquina. La aplicación recibe como parámetro de entrada el nombre de la imagen que se desea buscar.

---

<sup>85</sup><https://github.com/mmorejon/erase-una-vez-docker/tree/main/app>

La aplicación ha sido desarrollada en [Golang<sup>86</sup>](#) utilizando la dependencia [docker-client<sup>87</sup>](#). De forma predeterminada busca todas las imágenes de Nginx, pero este valor puede ser modificado a través del parámetro `--image-name`. Liste los ficheros de la carpeta `app` para ir conociendo los detalles del sistema.

```
$ tree app  
  
app  
├── Dockerfile-cache  
├── Dockerfile-dep  
├── Dockerfile-multi-stage  
├── go.mod  
├── go.sum  
└── main.go  
  
0 directories, 6 files
```

Si está familiarizado con las aplicaciones en *Golang* va a poder reconocer la estructura, en caso contrario le comento a grandes rasgos cada elemento. El fichero `main.go` tiene la lógica de la aplicación, mientras que los ficheros `go.mod` y `go.sum` registran las dependencias utilizadas en el código. Un homólogo a estos ficheros `go.*` son: `requirements.txt` en *Python*, `package.json` en *Node* o `composer.json` en *PHP*. Para construir la aplicación el primer paso tiene que ser la instalación de dependencias, y después se compila el código. En la carpeta `app` se ha incluido el fichero `Dockerfile-dep` con los pasos para construir la aplicación.

```
$ cat app/Dockerfile-dep  
  
# base image  
FROM golang:1.18.1-alpine3.15  
# set up work directory  
WORKDIR /opt/app  
# copy app files  
COPY . .  
# install dependencies  
RUN go mod download && \  
    go mod verify  
# build app  
RUN CGO_ENABLED=0 GOOS=linux go build -o /go/bin/app .  
# define run command  
ENTRYPOINT [ "app" ]
```

---

<sup>86</sup><https://go.dev/>

<sup>87</sup><https://pkg.go.dev/github.com/docker/docker/client>

En el fichero aparecen instrucciones nuevas que deben ser comentadas para entender su funcionamiento. También han sido incluidos múltiples comentarios antes de cada instrucción para apoyar en la comprensión del fichero.

La instrucción `WORKDIR`<sup>88</sup> establece el directorio de trabajo, si el directorio no existe, entonces se crea automáticamente. Todos los comandos a continuación del `WORKDIR` se ejecutan desde este directorio, su función es lo mismo a crear una carpeta y luego acceder a ella. Por otro lado, la instrucción `RUN`<sup>89</sup> es la ejecución de comando en consola, este elemento es muy utilizado en las imágenes para instalar dependencias y configurar sistemas. En la aplicación de ejemplo se utiliza para instalar las dependencias de *Golang* dentro de la imagen.

Puede utilizar la instrucción `RUN` tantas veces como sean necesarias, pero recuerde que cada instrucción es una nueva capa en su imagen, y mientras menos capas tenga mejor. Una forma de reducir el número de capas es utilizado el símbolo `&&` para unir múltiples comando en una ejecución.

Los pasos descritos en el fichero *Dockerfile-dep* se pueden resumir de la siguiente forma. Partiendo de *Golang* como base se crea el directorio `/opt/app`, luego se copian todos los ficheros desde directorio a esta carpeta, y se instalan las dependencias definidas en los ficheros `go.*`. Después se compila la aplicación, teniendo como resultado el binario `app`, que más tarde es configurado en el comando de arranque del contenedor.

Una vez conocido el contenido del fichero *Dockerfile-dep*, el próximo paso es construir la imagen. En el comando se ha introducido el parámetro `--file`, esta configuración permite indicarle a Docker el fichero de construcción de la imagen.

```
$ docker image build \
  --tag docker-book-app \
  --file app/Dockerfile-dep \
  app
...
=> => naming to docker.io/library/docker-book-app
```

El directorio `app` no cuenta con un fichero *Dockerfile*. El nombre *Dockerfile* es un acuerdo establecido por Docker como nombre predeterminado, pero usted puede darle el nombre que desee a su fichero. La recomendación es utilizar *Dockerfile* para el nombre, pero el ejercicio se ha diseñado de esta manera para mostrar el uso el parámetro `--file`.

El parámetro `--file` dentro del comando `build` define la ubicación del fichero de construcción de la imagen. La ruta toma como referencia el directorio donde se ejecuta el comando.

Una vez creada la imagen, es momento de iniciar el contenedor para conocer el resultado. Dentro del contenedor se va a intentar listar las imágenes de Docker, pero como no existe Docker dentro del contenedor, va ser necesario compartir el *socket* utilizado por *dockerd* en la máquina para comunicarse con el cliente *docker*. Recuerde la [sección de Arquitectura](#) vista al inicio del libro.

<sup>88</sup><https://docs.docker.com/engine/reference/builder/#workdir>

<sup>89</sup><https://docs.docker.com/engine/reference/builder/#run>

```
$ docker container run \
-v /var/run/docker.sock:/var/run/docker.sock \
docker-book-app
```

```
nginx:1.21.6
nginx:1.20.2
```

El resultado de la ejecución muestra las imágenes Nginx presentes en su ordenador, en su caso puede variar dependiendo de cuántas imágenes de Nginx tenga descargadas. El parámetro `-v` o `--volume` es utilizado para montar los ficheros, su objetivo es el mismo que el del parámetro `--mount`, pero utilizando una sintaxis simplificada. La siguiente línea corresponde con el mismo ejemplo, pero utilizando el comando `--mount`.

```
--mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock
```

Aproveche para jugar un poco con la aplicación. Inicie otro contenedor, pero con la intención de listar las imágenes *alpine*.

```
$ docker container run \
-v /var/run/docker.sock:/var/run/docker.sock \
docker-book-app \
--image-name alpine

alpine:latest
alpine:3.15.1
alpine:3.15.0
```

Los resultados en su máquina va a variar en dependencia de las imágenes que tenga descargadas. Los conceptos vistos en esta sección van a serle muy útiles para los proyectos con dependencias en los que trabaje. Dedique unos minutos a analizar cómo puede aplicar estos conocimientos en su entorno.

## Ganando velocidad utilizando la caché

Durante el proceso de desarrollo una aplicación se construye cientos o miles de veces, ya sea para realizar pruebas o para analizar su comportamiento. Ahora que su sistema vive dentro de un contenedor, posiblemente construya la imagen más o menos el mismo número de veces para confirmar que todo va como se espera.

Construir imágenes de contenedores puede parecer un proceso simple, pero si no tiene en cuenta algunos detalles el tiempo empleado superará las expectativas. Uno de estos detalles es la velocidad de construcción, la cual puede mejorar mucho haciendo uso de la caché de capas.

Las instrucciones del fichero *Dockerfile* que implican cambios en la estructura de ficheros provocan la generación de nuevas capas. Docker registra los cambios de las capas creadas, y luego estos cambios

son tomados como referencia para las próximas construcciones. Para ganar en velocidad durante el proceso de construcción Docker evalúa si la capa a crear es igual a la obtenida en la iteración anterior, si detecta que van a ser iguales, no la construye y utiliza la que tiene guardada (*en caché*).

Para detectar los cambios Docker analiza los ficheros de origen y el fichero *Dockerfile*. Cualquier cambio en uno de estos puntos descarta el uso de la caché. Además, Docker bloquea el uso de la caché para todas las instrucciones que vengan a continuación de una capa que haya tenido que ser construida.

Para entender mejor la situación planteada realice el ejercicio de construir la imagen de la aplicación anterior múltiples veces. La primera vez va a demorar un poco, pero luego las siguientes iteraciones son mucho más rápidas gracias a la caché. Fíjese en las líneas de salida marcadas con la palabra *CACHED*.

```
$ docker image build \
  --tag docker-book-app \
  --file app/Dockerfile-dep \
  app
...
=> CACHED [2/5] WORKDIR /opt/app
=> CACHED [3/5] COPY . .
=> CACHED [4/5] RUN go mod download &&      go mod verify
=> CACHED [5/5] RUN GOOS=linux go build -o /go/bin/app .
...
```

Ahora realice una simple modificación en el fichero *main.go*. Por ejemplo, en la línea 15 cambie la palabra *nginx* por *alpine*, luego vuelva a construir la imagen.

```
$ docker image build \
  --tag docker-book-app \
  --file app/Dockerfile-dep \
  app
...
=> CACHED [2/5] WORKDIR /opt/app
=> [3/5] COPY . .
...
```

Note que se ha eliminado la marca *CACHED* en el tercer paso, y de ahí en adelante todos los pasos tuvieron que ser ejecutados nuevamente. Al realizar la copia de los ficheros Docker detectó cambios, por tal motivo los pasos siguientes tuvieron que ser ejecutados. Intente ahora dar respuesta a las siguientes preguntas. *¿Será necesario instalar las dependencias nuevamente si el cambio fue en el fichero main.go? ¿Cómo se puede evitar la instalación de dependencias constantemente?*

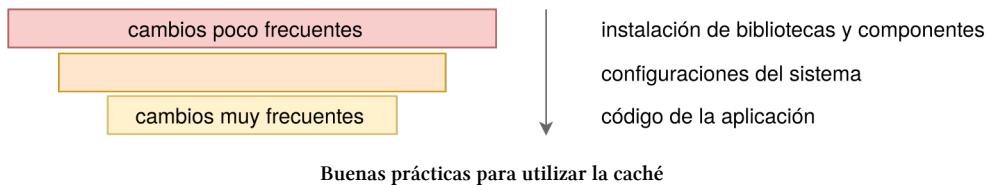
La respuesta a la primera pregunta es *No*. Los ficheros *go.\** son los encargados de gestionar las dependencias, si no hay cambios en estos ficheros no es necesario volver a instalarlas. Para evitar

la instalación de dependencias de forma constante hay que modificar el fichero *Dockerfile-dep*. El fichero *Dockerfile-cache* contiene los cambios necesarios para sacar provecho de la caché y mejorar la velocidad.

```
$ cat app/Dockerfile-cache

# base image
FROM golang:1.18.1-alpine3.15
# set up work directory
WORKDIR /opt/app
# copy files dependencies
COPY go.* /opt/app
# install dependencies
RUN go mod download && \
    go mod verify
# copy app files
COPY . .
# build app
RUN CGO_ENABLED=0 GOOS=linux go build -o /go/bin/app .
# define run command
ENTRYPOINT [ "app" ]
```

La principal diferencia está en la copia e instalación de dependencias antes de copiar el código fuente de la aplicación. De esta forma se garantiza otra buena práctica para el fichero *Dockerfile*, donde los cambios menos frecuentes deben realizarse primero, quedando para el final los cambios más frecuentes, como por ejemplo, los cambios en el código de la aplicación.



Construya nuevamente la imagen, pero esta vez utilizando el fichero *Dockerfile-cache*.

```
$ docker image build \
  --tag docker-book-app \
  --file app/Dockerfile-cache \
  app
...
=> => naming to docker.io/library/docker-book-app
```

Modifique el fichero `main.go` y construya nuevamente la imagen. Note el uso de la caché hasta el paso previo de la creación del binario.

```
$ docker image build \
--tag docker-book-app \
--file app/Dockerfile-cache \
app

=> CACHED [4/6] RUN go mod download &&      go mod verify
=> [5/6] COPY . .
=> [6/6] RUN GOOS=linux go build -o /go/bin/app .
```

Los grupos de desarrollo utilizan flujos de integración y despliegues continuos (*CI/CD*). En cada iteración se construye una imagen para ser desplegada en los servidores. Teniendo esto en mente, es de vital importancia diseñar correctamente los ficheros *Dockerfiles* para sacar provecho de la caché siempre que sea posible. El uso de las buenas prácticas explicadas en esta sección le serán de mucha ayuda en estos escenarios.

## Construir para múltiples plataformas

Los contenedores se han extendido a todos los sectores tecnológicos. Es común verlos en plataformas consolidadas en el mercado como AWS, GCloud y Azure, pero también en laboratorios construidos con RaspberryPi y en dispositivos IoT. Esto brinda una gran ventaja porque se puede programar una vez y luego ejecutarse en cualquier plataforma sin mayores dificultades. Sin embargo, esto no sucede por arte de magia, es necesario tener en cuenta algunos detalles para garantizar el correcto funcionamiento de los contenedores en múltiples arquitecturas.

Lo primero va a ser conocer las arquitecturas de *hardware* donde puede funcionar un contenedor con Docker. En el listado se encuentran por ejemplo *amd64*, *arm64v8*, *arm32v7* y *windows-amd64*, pero le recomiendo acceder a la [documentación oficial<sup>90</sup>](#) para conocer el listado completo.

La arquitectura de una imagen se define durante su construcción utilizando el parámetro `platform`. Si este parámetro no se especifica, la arquitectura de la imagen va a ser la misma de su ordenador. Utilice los siguientes comandos para comprobar la teoría.

Conozca la arquitectura de su ordenador. Los resultados de los siguientes van a cambiar según el ordenador que usted tenga.

```
$ docker info --format '{{.OSType}}/{{.Architecture}}'

linux/aarch64
```

Ahora consulte la arquitectura de la imagen creada en el ejercicio anterior. El resultado va a corresponder con el valor mostrado para su ordenador. En este ejemplo `aarch64` es lo mismo que `arm64`. Ver [enlace<sup>91</sup>](#) para más información.

<sup>90</sup><https://github.com/docker-library/official-images#multiple-architectures>

<sup>91</sup><https://en.wikipedia.org/wiki/AArch64>

```
$ docker image inspect docker-book-app --format '{{.Os}}/{{.Architecture}}'  
linux/arm64
```

Hasta aquí todo va bien, porque usted puede iniciar el contenedor en su ordenador sin contratiempos, pero *¿qué sucedería si intenta desplegar su servicio en una máquina con arquitectura distinta a la suya? ¿Podrá funcionar correctamente?*

Seguramente se imagina la respuesta, ese contenedor no va a poder funcionar en una arquitectura distinta. En aras de buscar una solución al problema, Docker ha incluido la capacidad de especificar la plataforma para la que se desea construir la imagen a través del parámetro `--platform`. El ejemplo que aparece a continuación pretende construir la misma imagen, pero cambiando su arquitectura. En su caso utilice una plataforma distinta a la existente en su ordenador

```
$ docker image build \  
  --tag docker-book-app \  
  --file app/Dockerfile-cache \  
  --platform=linux/amd64 \  
  app  
...  
=> => naming to docker.io/library/docker-book-app
```

Si vuelve a preguntar por la arquitectura de la imagen compruebe que ha cambiado de `linux/arm64` a pasado a ser `linux/amd64`.

```
$ docker image inspect docker-book-app --format '{{.Os}}/{{.Architecture}}'  
linux/amd64
```

Ahora intente iniciar un contenedor a partir de esta imagen que tiene una arquitectura diferente a su ordenador.

```
$ docker container run \  
  -v /var/run/docker.sock:/var/run/docker.sock \  
  docker-book-app
```

```
WARNING: The requested image's platform (linux/amd64) does not match the detected host pl\  
atform (linux/arm64/v8) and no specific platform was requested  
nginx:1.21.6  
nginx:1.20.2
```

El resultado mostrado en consola corresponde con el esperado. Para este ejemplo el cambio de arquitectura no dificultó el inicio del servicio, esto se debe a que el binario fue compilado para linux,

pero para el resto de aplicaciones debe tener precaución con el funcionamiento. Docker por su parte, imprime siempre un mensaje de advertencia cuando detecta la diferentes arquitecturas entre el *host* y la imagen.

La imagen *docker-book-app* cambió de plataforma al cambiar el comando de construcción. Cada vez que necesite cambiar el valor de la plataforma debe construir una nueva imagen, lo cual puede llegar a ser tedioso y repetitivo en muchas ocasiones, más cuando son muchos los valores de plataformas posibles a establecer.

*¿Será que existe una forma fácil de construir una imagen para múltiples plataformas al mismo tiempo?*

Sí existe la forma, de hecho, los equipos construyen y almacenan las imágenes para todas las plataformas que necesiten en un solo paso. Para lograr la configuración mencionada se necesitan conocimientos sobre cómo publicar imágenes en registros de Docker. Este contenido se va a explicar en el próximo capítulo, por lo tanto debe tener un poco de paciencia para saber cómo hacerlo.

Aproveche para acceder nuevamente a la imagen [hello-world en Docker Hub<sup>92</sup>](#) y conozca todas las plataformas para las que ha sido construida la imagen.

## Utilizar estructura multi-stage en los Dockerfiles

Son múltiples y diversos los temas asociados a la construcción de imágenes, pero realmente hay bastantes más. La recomendación es ir paso a paso para asimilar los conceptos fundamentales, luego va a poder profundizar en aquellos que sean de su preferencia.

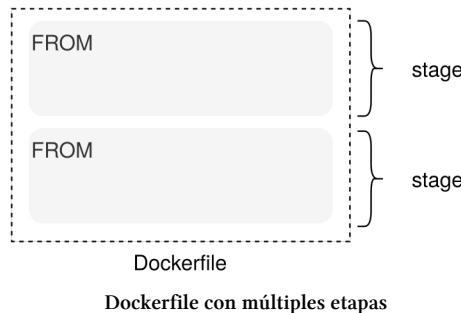
Un último elemento a ver antes de cerrar el capítulo son los múltiples etapas en los ficheros *Dockerfiles*, en inglés puede encontrar este tema con el término *multi-stages*. El uso de múltiples etapas le va a permitir crear imágenes pequeñas sin importar lo complejo o largo que sea su proceso de construcción.

La construcción de las aplicaciones demandan muchas dependencias para compilar, comprimir o minimizar el código. Sin embargo, nada de esto va a ser necesario para su funcionamiento. Mientras más pasos de configuración o instalación tenga su *Dockerfile*, más grande va a ser la imagen final, pero ya sabe que esto no es bueno. Aquí es donde aparecen ambos lados de la moneda, se necesitan bases de imágenes grandes con múltiples herramientas para poder compilar el código, pero se recomiendan imágenes pequeñas y ligeras para ejecutar el código compilado en producción. Para tener lo bueno de ambos mundos va a ser necesario utilizar *multi-stages*.

Las etapas en el *Dockerfile* son bloques de instrucciones delimitadas por la instrucción *FROM*. El contenido generado en cada etapa puede ser compartido con el resto a través de la instrucción *COPY*.

---

<sup>92</sup>[https://hub.docker.com/\\_/hello-world](https://hub.docker.com/_/hello-world)



## Definir múltiples etapas

La instrucción *FROM* define el inicio de una etapa en el *Dockerfile*. Los ficheros vistos hasta el momento tienen una sola instrucción *FROM*, pero puede incluir todas las que desee. El ejemplo que se muestra a continuación corresponde con el contenido del fichero *Dockerfile-multi-stage* del repositorio de ejemplos en la carpeta *app*.

```
# base image to build and compile the code
FROM golang:1.18.1-alpine3.15 as builder
...
...
COPY .
# build app
RUN CGO_ENABLED=0 GOOS=linux go build -o /go/bin/app .

# base image to run the app
FROM scratch
COPY --from=builder /go/bin/app /app
ENTRYPOINT ["/app"]
```

Se han ocultado algunas líneas para resaltar el contenido relevante para la sección. Preste atención a las dos etapas definidas, en la primera se copia el código, se instalan las dependencias y se compila la aplicación. En la segunda etapa, se utiliza la imagen *scratch* como base y se copia el binario generado en la etapa inicial.

Cada etapa tiene una numeración automática según el orden de aparición, pero también pueden identificarse a través de alias. El objetivo de ambos mecanismos es crear referencias que puedan ser utilizadas luego en otras etapas del *Dockerfile*. En el ejemplo se ha marcado la primera etapa con el alias *builder*, luego este alias se utiliza en la etapa final, utilizando el parámetro *--from* con el fin de copiar el binario hacia la imagen final.

## Construcción utilizando múltiples etapas

¿Cómo se construye un *Dockerfile multi-stage*?

Docker recorre el fichero *Dockerfile* de inicio a fin, ejecutando las instrucciones y generando las capas que sean necesarias. Sin embargo, la imagen final va a ser la creada en la última etapa. Construya una nueva imagen utilizando el fichero *Dockerfile-multi-stage* para comprobarlo.

```
$ docker image build \
--tag docker-book-app \
--file app/Dockerfile-multi-stage \
app
```

Va a ver todas las instrucciones del *Dockerfile-multi-stage* ejecutadas, pero en la historia solamente están reflejados los pasos de la última etapa.

```
$ docker image history docker-book-app \
--format 'table {{.ID}}\t{{.CreatedBy}}\t{{.Size}}'
```

IMAGE	CREATED BY	SIZE
6c59d95fd132	ENTRYPOINT ["/app"]	0B
<missing>	COPY /go/bin/app /app # buildkit	11.6MB

El resto de capas construidas se quedan almacenadas en la máquina para ser utilizadas por la caché, pero nunca llegan a formar parte de la imagen final.

Aunque este sea el flujo más utilizado, también existe la posibilidad de especificar la construcción de una etapa a través del parámetro `--target`. Este parámetro olvida completamente el flujo anterior y construye una imagen con los pasos presentes en la etapa especificada. El resto de instrucciones del fichero son ignoradas.

Utilice el siguiente comando para ver un ejemplo con este parámetro. Las instrucciones de la última etapa van a ser ignoradas.

```
$ docker image build \
--tag docker-book-app \
--file app/Dockerfile-multi-stage \
--target=builder app
```

Compruebe las capas creadas para esta imagen.

```
$ docker image history docker-book-app \
--format 'table {{.ID}}\t{{.CreatedBy}}\t{{.Size}}'

IMAGE          CREATED BY                               SIZE
f5774fd02b44  ENTRYPPOINT ["app"]
<missing>      RUN /bin/sh -c GOOS=linux go build -o /go/bi... 44.1MB
<missing>      COPY . . # buildkit                           23.1kB
<missing>      RUN /bin/sh -c go mod download &&       go mod... 206MB
<missing>      COPY go.* /opt/app # buildkit                  21.3kB
<missing>      WORKDIR /opt/app                            0B
<missing>      /bin/sh -c #(nop) WORKDIR /go                   0B
<missing>      /bin/sh -c mkdir -p "$GOPATH/src" "$GOPATH/b... 0B
...
...
```

La construcción de etapas de forma individual son de mucha utilidad cuando desee comprobar el comportamiento de una sección específica, para el resto de casos, le recomiendo mantener el flujo predeterminado.

## Imágenes finales de referencia

Ahora que domina la técnica *multi-stage* le recomiendo revisar las imágenes base más utilizadas para los estados finales. No deben ser cualquier tipo de imagen, deben ser pequeñas y con la mínima cantidad de herramientas instaladas. Estas características van a garantizar un rápido movimiento de la imagen a través de los servidores, así como un alto nivel de seguridad con relación a las vulnerabilidades.

Como recomendación le propongo utilizar, siempre que sea posible, las siguientes imágenes como base para su etapa final.

[Scratch<sup>93</sup>](#): aquí no existen ficheros previos, es empezar desde cero. Las aplicaciones que puedan ser compiladas y que no tengan grandes dependencias pueden hacer uso de esta base.

[Alpine<sup>94</sup>](#): es una distribución de Linux muy pequeña (5 MB). Cuenta con interesantes bondades a la hora de instalar herramientas.

[Distroless<sup>95</sup>](#): son imágenes pequeñas creadas y mantenidas por el equipo de Google, donde han incluido las buenas prácticas sobre seguridad en contenedores.

Estas son las recomendaciones para las imágenes base, pero si no las utiliza no significa que esté mal, lo importante es mantener la imagen lo más pequeña posible.

---

<sup>93</sup>[https://hub.docker.com/\\_/scratch](https://hub.docker.com/_/scratch)

<sup>94</sup>[https://hub.docker.com/\\_/alpine](https://hub.docker.com/_/alpine)

<sup>95</sup><https://github.com/GoogleContainerTools/distroless>

## Ejercicios para practicar lo aprendido

Después de haber incorporado nuevos conceptos y comandos es momento de practicar. Los ejercicios que aparecen a continuación tienen la intención de pasar un rato agradable delante del Terminal.

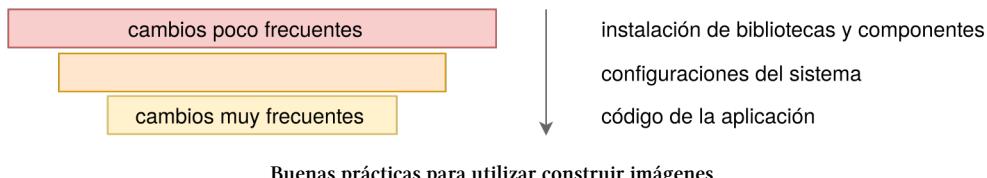
1. ¿Cuál de las siguientes imágenes tiene mayor número de capas? *nginx:1.21.6* o *redis:6.2.6*.
2. ¿Cuántos ficheros *Dockerfiles* puede tener una aplicación? y ¿Cuál es la recomendación al respecto?
3. ¿Qué problema resuelve el fichero *.dockerignore*?
4. Utilice la imagen *nginx:1.21.6* para construir su propio sitio web. Cree una página *html* con los datos de su currículum, luego cree su propia imagen e inicie un contenedor utilizando el puerto 8000 de su máquina.
5. Utilice la imagen *alpine:3.15.1* como base para instalar *curl*. Luego cambie el comando *ENTRYPOINT* para que utilice la nueva herramienta instalada. Utilice un fichero *Dockerfile* para dejar definidas las instrucciones.
6. Construya una aplicación sencilla que muestre un mensaje en pantalla con el lenguaje de su preferencia. Luego adicione el fichero *Dockerfile* para generar la imagen que permita poner en funcionamiento la aplicación. Suba estos ficheros a un repositorio en GitHub y comparta el enlace con el autor para recibir sus comentarios.

## Resumen del capítulo

Este ha sido uno de los capítulos más largos del libro porque hay mucho contenido relacionado con la construcción de imágenes. Las secciones vistas anteriormente constituyen el primer paso a dar, pero le recomiendo explorar otros contenidos relacionados el tema.

Será importante recordar que las imágenes están compuestas por capas (*layers*), donde cada capa significa que han existido cambios en la estructura de ficheros. También está el fichero *Dockerfile*, lugar donde se describe como código la construcción de la imagen. Pueden existir muchos ficheros *Dockerfiles* en un repositorio, pero en la mayoría de los casos con uno es suficiente.

Las imágenes construidas deben incluir aquellos componentes indispensables para que el sistema o la aplicación funcione. Mientras más pequeñas sean las imágenes más rápido va a ser su proceso de construcción, también será menos costoso su almacenamiento. Utilice la técnica *multi-stage* para quedarte solamente con aquellos ficheros que sean indispensables para la aplicación.



Los conceptos aprendidos en el capítulo le permiten construir imágenes, pero el viaje no termina aquí, aún quedan preguntas por resolver, como por ejemplo:

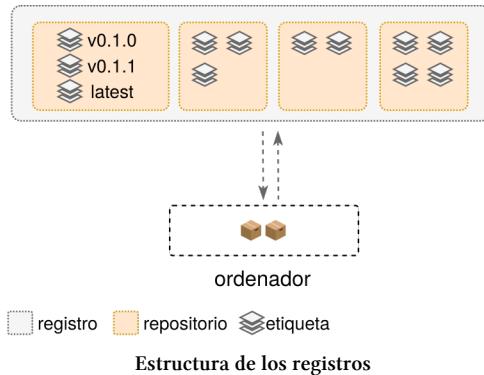
- *¿Dónde deben ser almacenadas las imágenes? ¿En la máquina local o en la nube?*
- *¿Cuántos registros públicos de imágenes se pueden utilizar?*
- *¿Cómo mover una imagen desde la máquina hacia las plataformas en la nube?*

¡Si desea dar respuesta a estas preguntas lea el próximo capítulo!

# Publicar imágenes

Los tres verbos que caracterizan a Docker son: construir, distribuir y desplegar. En el capítulo anterior se explicó cómo construir una imagen, por lo tanto, le toca el turno al segundo verbo: *distribuir*.

Las aplicaciones son empaquetadas en imágenes, pero luego deben ser movidas hacia algún lugar fuera de su ordenador. El objetivo de esta acción es habilitar su acceso desde cualquier plataforma. Estos almacenes de imágenes donde se alojan las aplicaciones se llaman Registros. Cada Registro puede contener miles de repositorios, y cada repositorio contiene múltiples etiquetas de la misma imagen.



Actualmente casi todos los proveedores de infraestructura cuentan con un Registro para las imágenes de contenedores. Realice una consulta en la plataforma que más utilice y es casi seguro que va a encontrar un Registro para almacenar imágenes. El objetivo que persiguen los proveedores de infraestructura es permitirle al usuario almacenar sus aplicaciones lo más cerca posible del lugar donde van a ser desplegadas, de esta forma reducen el tiempo de descarga y el coste de tráfico de paquetes.

A esta competencia se han sumado las plataformas de código como [GitLab<sup>96</sup>](#) y [GitHub<sup>97</sup>](#), pero en este caso con la intención de almacenar su aplicación lo más cerca posible del código fuente. Esta filosofía tiene mucha aceptación desde dos puntos de vista, el primero es para las aplicaciones open-source, donde su objetivo no es desplegar el sistema, sino hacerlo accesible al resto de la comunidad. El segundo elemento es para los equipos donde se utilizan múltiples plataformas, porque evitan tener la aplicación replicada por los registros de cada una de ellas.

A la fiesta de Registros se suma [Docker Hub<sup>98</sup>](#), que ha sido el primero de todos los registros creados en la industria y cuenta con un gran número de funcionalidades e integraciones, tanto para equipos

<sup>96</sup><https://about.gitlab.com/>

<sup>97</sup><https://github.com/>

<sup>98</sup><https://hub.docker.com/>

pequeños, como para grandes empresas. Docker Hub también comparte la filosofía de tener un único lugar para todas las imágenes, luego estas pueden ser desplegadas en la infraestructura que deseé.

Existen muchos otros Registros, con un poco de tiempo va a poder explorarlos y hacerse con un criterio propio de cuál puede ser mejor para sus intereses. A la hora de analizar o evaluar un Registro le recomiendo tener en cuenta los siguientes aspectos:

- Costes del almacenamiento.
- Número de imágenes públicas y privadas permitidas.
- Escáner de vulnerabilidades.
- Integraciones con otras plataformas y servicios.

Después de esta breve introducción debe haber notado la importancia del tema a tratar, porque en algún momento va a necesitar almacenar sus imágenes fuera del ordenador. Es por este motivo que se ha creado el capítulo, para mostrar cómo publicar las imágenes de contenedores e interactuar con los Registros.

## Publique su primera imagen en ttl.sh

Para publicar una imagen el primer paso es tener la imagen creada. El comando para crear la imagen fue visto en el capítulo anterior, pero no está de más recordarlo. Los comandos que aparecen a continuación tienen que ser ejecutados desde la raíz del [repositorio de ejercicios](#)<sup>99</sup>.

```
$ docker image build \
  --tag docker-book-app \
  --file app/Dockerfile-multi-stage \
  ./app/
```

Luego compruebe que la imagen se ha creado correctamente.

```
$ docker image list docker-book-app
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-book-app	latest	426a0e2d3db5	13 seconds ago	12MB

¡Perfecto! La imagen se ha creado correctamente. El siguiente paso va a ser mover la imagen hacia un Registro en Internet.

---

<sup>99</sup><https://github.com/mmorejon/erase-una-vez-docker>

## Registro de imágenes ttl.sh

El primer registro a utilizar va a ser [ttl.sh](https://ttl.sh/)<sup>100</sup>. TTL es un proyecto open-source fácil de utilizar donde no necesita tener cuenta de usuario ni credenciales para publicar o descargar las imágenes. Las imágenes en este registro son efímeras, cada imagen puede estar almacenada como máximo 24 horas, después de este tiempo se eliminan automáticamente.

El único requisito que pone el registro es el nombre de la imagen, el cual tiene que seguir el siguiente formato: `ttl.sh/<uuid>`. El [UUID](#)<sup>101</sup> es un identificador utilizado para garantizar que cada imagen sea única. En este momento la imagen que deseamos publicar no cumple con el formato establecido por el registro TTL, por lo tanto hay que buscar la forma de modificarlo.

Docker permite establecer nombres a las imágenes a través del comando `image tag`. Esta funcionalidad es muy útil cuando se necesita publicar la imagen en registros distintos; recuerde que cada registro tiene una estructura de nombre distinta al resto. El siguiente ejemplo muestra cómo utilizar este nuevo comando.

```
$ IMAGE_NAME=$(uuidgen | tr "[[:upper:]]" "[[:lower:]]")
$ docker image tag docker-book-app ttl.sh/${IMAGE_NAME}
```

Ahora liste nuevamente las imágenes y compruebe que existen dos referencias apuntando al mismo contenido. El comando mostrado a continuación filtra el listado de imágenes utilizando como referencia el nombre. Fíjese que el identificador de las imágenes es el mismo, porque es la misma imagen pero con dos referencias.

```
$ docker image ls \
  --filter=reference=docker-book-app \
  --filter=reference="ttl.sh/*" \
  --format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}'

REPOSITORY          TAG      IMAGE ID
docker-book-app    latest   426a0e2d3db5
ttl.sh/5195e4c3-50a1-42d5-97ce-12b08e55d4d9  latest   426a0e2d3db5
```

La imagen ha quedado etiquetada con la estructura de nombre esperada, ahora es posible publicarla en el registro TTL. El comando que se utiliza para publicar las imágenes es `image push`, y como parámetro se pasa el nombre completo que aparece en la columna repositorio junto con la etiqueta.

---

<sup>100</sup><https://ttl.sh/>

<sup>101</sup>[https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier#Version\\_4\\_.28random.29](https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_.28random.29)

```
$ docker image push ttl.sh/${IMAGE_NAME}

The push refers to repository [ttl.sh/5195e4c3-50a1-42d5-97ce-12b08e55d4d9]
eb115869c6c8: Pushed
latest: digest: sha256:76c2814df73494b6ba2e457d46382b75a9911966438b0d2971f4af5df7a450d6 s\
ize: 527
```

La subida de la imagen es muy rápida porque apenas pesa 12 megabyte, de aquí la importancia de construir imágenes ligeras. Los mensajes en la salida del comando muestran que ha sido copiada correctamente. En este momento la imagen se encuentra en su máquina, pero también en el registro público en Internet.

Antes de cumplirse las 24 horas de almacenamiento reservadas en TTL, realice el siguiente experimento: elimine todas las imágenes locales y luego descargue la imagen almacenada en el registro público. Este sería el mismo caso si intenta desplegar la imagen en un servicio de contenedores en las infraestructuras públicas.

Utilice el comando `image prune` para eliminar todas las imágenes almacenadas en su ordenador. También deben ser eliminadas las capas intermedias creadas durante la construcción de imágenes, estas últimas pueden ser eliminadas a través del comando `builder prune`. En ambos comandos se ha utilizado el parámetro `--force` para evitar la pregunta de confirmación sobre la eliminación de los datos.

```
$ docker image prune --all --force && \
  docker builder prune --force
```

Liste las imágenes para estar seguros que el resultado es vacío. No deben haber imágenes en su ordenador en este momento.

```
$ docker image ls
```

Luego utilice el comando `image pull` para descargar la imagen almacenada en el registro TTL. Docker va a buscar la imagen en su ordenador, pero al no estar presente, tiene que ir al registro para descargarla.

```
$ docker image pull ttl.sh/$IMAGE_NAME

Using default tag: latest
latest: Pulling from 5195e4c3-50a1-42d5-97ce-12b08e55d4d9
137100dc60f4: Pull complete
Digest: sha256:76c2814df73494b6ba2e457d46382b75a9911966438b0d2971f4af5df7a450d6
Status: Downloaded newer image for ttl.sh/5195e4c3-50a1-42d5-97ce-12b08e55d4d9:latest
ttl.sh/5195e4c3-50a1-42d5-97ce-12b08e55d4d9:latest
```

Liste las imágenes para confirmar la existencia del elemento descargado.

```
$ docker image ls \
--format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}'

REPOSITORY           TAG      IMAGE ID
ttl.sh/5195e4c3-50a1-42d5-97ce-12b08e55d4d9  latest   426a0e2d3db5
```

¡Muy bien! Ha traído de vuelta la imagen con la aplicación. Aproveche este momento para iniciar un contenedor y así verificar que funciona correctamente. El resultado de la ejecución va a ser vacío porque no existen imágenes con referencia `nginx` en el ordenador.

```
$ docker container run \
-v /var/run/docker.sock:/var/run/docker.sock \
ttl.sh/$IMAGE_NAME
```

There are no images with the reference: nginx

Como puede comprobar, la imagen descargada funciona exactamente igual a la construida en su ordenador. No existen cambios ni modificaciones en las imágenes durante su almacenamiento en los registros de imágenes.

¡Enhorabuena por este nuevo paso! Ahora usted puede compartir sus aplicaciones en forma de imagen de contenedores con el resto del mundo.

## Gestionar múltiples etiquetas por imagen

Las etiquetas son las encargadas de darle un nombre a las imágenes. Este nombre permite identificar cada imagen del resto, pero también establece el registro donde va a ser almacenada. Las imágenes pueden tener múltiples etiquetas, y estas etiquetas pueden especificarse en diferentes momentos durante el flujo de trabajo.

La forma rápida de establecer múltiples etiquetas a una imagen es durante su construcción. El comando `image build` permite establecer todas las etiquetas que necesite a través del parámetro `--tag`. El siguiente ejemplo muestra cómo queda el comando, la variable `IMAGE_NAME` contiene el mismo valor `UUID` utilizado en la sección anterior.

```
$ docker image build \
--tag docker-book-app \
--tag ttl.sh/${IMAGE_NAME} \
--file app/Dockerfile-multi-stage \
./app/
```

Como puede apreciar, el parámetro `--tag` va a estar presente tantas veces como sea necesario. Al terminar la ejecución del comando va a tener dos etiquetas haciendo referencia a la misma imagen. Luego puede empujar la imagen hacia el registro en un segundo paso de forma rápida y sencilla.

Si desconoce la estructura de nombre del registro puede poner un valor temporal, porque va a poder establecer la etiqueta correcta en otro momento a través del comando `image tag`. Si tiene dudas sobre cómo utilizar este comando haga uso del parámetro `--help` para recordar el orden de los elementos.

```
$ docker image tag --help

Usage: docker image tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]

Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```

El siguiente ejemplo muestra como adicionar una nueva etiqueta a la imagen `docker-book-app`.

```
$ docker image tag docker-book-app my-new-tag
```

Liste las imágenes y compruebe que la nueva etiqueta ha sido asignada correctamente.

```
$ docker image ls \
--filter=reference=my-new-tag \
--filter=reference=docker-book-app \
--format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}'

REPOSITORY      TAG      IMAGE ID
docker-book-app  latest   973a3a98fe1a
my-new-tag       latest   973a3a98fe1a
```

¡Genial! Ambas etiquetas están como se esperaba. Ahora va a poder adicionar las referencias que desee a sus imágenes.

Hasta el momento se han establecido nuevas etiquetas a las imágenes, pero también hace falta saber cómo eliminarlas en caso de haber cometido algún error en el nombre. Las etiquetas en las imágenes no se editan porque son referencias de nombre hacia una imagen, por lo tanto solamente pueden adicionararse y eliminarse.

Para eliminar una etiqueta se utiliza el comando `image rm`, el cual es responsable de eliminar las imágenes, pero en este caso hay múltiples etiquetas apuntando a la misma imagen, por lo tanto se

va a eliminar la referencia deseada y la imagen seguirá existiendo en su ordenador. Este comando elimina la imagen completamente cuando no queden referencias apuntando a la imagen.

Realice el ejercicio de eliminar la etiqueta `my-new-tag` para poner en práctica este nuevo concepto.

```
$ docker image rm my-new-tag
```

```
Untagged: my-new-tag:latest
```

Como puede observar, el mensaje en la consola indica que ha eliminado la etiqueta `my-new-tag` y no la imagen. Vuelva a listar las imágenes para comprobar que la etiqueta ha sido eliminada.

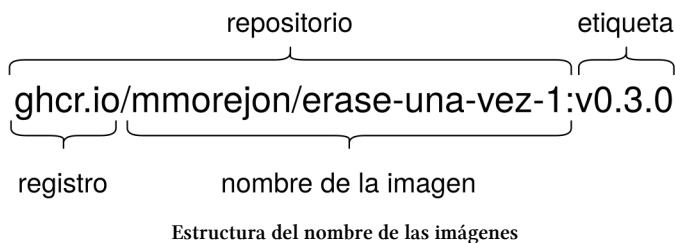
```
$ docker image ls \
--filter=reference=my-new-tag \
--filter=reference=docker-book-app \
--format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}'
```

REPOSITORY	TAG	IMAGE ID
docker-book-app	latest	973a3a98fe1a

¡Enhorabuena! En este punto usted es capaz de etiquetar las imágenes a su gusto. Este es un paso importante para poder publicar luego la imagen en el registro de su preferencia.

## Múltiples versiones por imagen

En los comandos anteriores se ha etiquetado la imagen haciendo referencia al nombre, pero recuerde que también puede especificar versiones en la imagen. Las versiones de las imágenes, al igual que en las aplicaciones, van a permitirle gestionar la evolución de cambios del sistema.



Utilice el comando `image tag`, pero en esta ocasión para establecer múltiples versiones de una misma imagen.

```
$ docker image tag docker-book-app docker-book-app:v0.1.0
$ docker image tag docker-book-app docker-book-app:v0.1.1
$ docker image tag docker-book-app docker-book-app:v0.2.0
```

Liste las imágenes asociadas a la referencia docker-book-app.

```
$ docker image ls \
--filter=reference=docker-book-app \
--format 'table {{.Repository}}\t{{.Tag}}\t{{.ID}}'
```

REPOSITORY	TAG	IMAGE ID
docker-book-app	latest	973a3a98fe1a
docker-book-app	v0.1.0	973a3a98fe1a
docker-book-app	v0.1.1	973a3a98fe1a
docker-book-app	v0.2.0	973a3a98fe1a

Todas las etiquetas hacen referencia a la misma imagen, pero en entornos reales cada versión va a corresponder con versiones distintas de la aplicación.

## Publicar imágenes en Docker Hub

Docker Hub<sup>102</sup> fue el primer registro de imágenes de contenedores en la industria, y todavía sigue siendo uno de los más utilizados. Esta es la razón por la cual vale la pena dedicar una sección para explicar algunos aspectos interesantes sobre su funcionamiento.

El primer paso para utilizar Docker Hub es crearse una cuenta en la plataforma. Los pasos para lograrlo son muy sencillos, [acceda a este enlace<sup>103</sup>](https://hub.docker.com/signup) y complete los campos requeridos. Siempre se recomienda activar el segundo factor de autenticación (2FA) para una mayor seguridad de la cuenta. A la hora de especificar su usuario busque un nombre fácil de escribir y sin caracteres complicados.

Una vez creada la cuenta le propongo dedicar unos minutos para navegar por las opciones disponibles en la plataforma. Acceda a su perfil para completar sus datos y configure las notificaciones para recibir solamente la información que sea de su interés.

## Docker Hub CLI

A través del navegador web puede acceder a todas las opciones en Docker Hub, sin embargo, explicar cada paso a través de capturas de pantallas no es la mejor opción. Para felicidad del autor, Docker ha creado la herramienta [hub-tool<sup>104</sup>](https://github.com/docker/hub-tool), la cual permite comunicarse con Docker Hub vía API. Esta herramienta va a permitir que ambos, lectores y autor, tengan la misma vista a la hora de gestionar

<sup>102</sup><https://hub.docker.com/>

<sup>103</sup><https://hub.docker.com/signup>

<sup>104</sup><https://github.com/docker/hub-tool>

los repositorios. El proyecto se encuentra en fase experimental, pero funciona bien para el objetivo buscado en esta sección.

Para hacer uso de `hub-tool` es necesario descargar el archivo comprimido con el binario, extraer su contenido y colocar el binario de forma tal que pueda ser ejecutado. La versión utilizada es la [v0.4.5<sup>105</sup>](#) y puede ser descargada directamente desde GitHub. Si tiene dudas sobre cómo instalar la herramienta puede apoyarse en la [documentación del propio repositorio<sup>106</sup>](#).

Después de instalar la herramienta confirme la versión instalada a través del siguiente comando.

```
$ hub-tool version  
Version: v0.4.5  
Git commit: d383e722fffd4f0c7ccf835aace7138abb2e937
```

Consulte las opciones disponibles utilizando el parámetro `--help`.

```
$ hub-tool --help  
A tool to manage your Docker Hub images
```

Usage:

```
hub-tool  
hub-tool [command]
```

Available Commands:

account	Manage your account
help	Help about any command
login	Login to the Hub
logout	Logout of the Hub
org	Manage organizations
repo	Manage repositories
tag	Manage tags
token	Manage Personal Access Tokens
version	Version information about this tool

El próximo paso es hacer login con la herramienta `hub-tool` para acceder con su usuario a Docker Hub. Si ha configurado el segundo factor de autenticación el comando le va a pedir el código de seis cifras para completar el registro, en caso contrario solamente va a necesitar su contraseña.

---

<sup>105</sup><https://github.com/docker/hub-tool/releases/tag/v0.4.5>

<sup>106</sup><https://github.com/docker/hub-tool#install>

```
$ hub-tool login <su-usuario>

Password:
2FA required, please provide the 6 digit code:
Login Succeeded
```

En este punto la herramienta se encuentra lista para su uso. Es momento entonces de avanzar a la próxima sección, donde va a poder crear su primer repositorio en Docker Hub.

## Crear repositorio y publicar la imagen

Los repositorios en Docker Hub pueden crearse de dos formas, ambas totalmente válidas según sean sus necesidades. La primera es a través del navegador web, donde puede escribir el nombre, la descripción del repositorio y la visibilidad (privado o público). Utilice el siguiente enlace para crear un nuevo repositorio. No olvide establecer su usuario al final de la *URL*.

<https://hub.docker.com/repository/create?namespace=su-usuario>

La segunda opción es crear el repositorio en el momento de publicar la imagen. Cuando se utiliza el comando `image push` hacia un repositorio en Docker Hub, la plataforma es capaz de crear el repositorio si este no existe. La visibilidad del repositorio es pública por defecto, a no ser que especifique lo contrario en las políticas de privacidad de su usuario en la plataforma.

Las instrucciones que aparecen a continuación le van a ayudar a crear su primer repositorio en Docker Hub.

Consulte los repositorios creados en su cuenta antes de crear uno nuevo, esta acción es importante porque los nombres de los repositorios no pueden repetirse. Utilice la herramienta `hub-tool` para listar los repositorios presentes en su cuenta.

```
$ hub-tool repo ls
```

El listado debe aparecer vacío si no ha trabajado con Docker Hub previamente, de lo contrario, van a aparecer los repositorios creados en momentos anteriores.

El próximo paso es referenciar correctamente la imagen a través de etiquetas. Docker Hub tiene su propia estructura de nombre para los repositorios, la cual debe ser respetada y establecida para garantizar el almacenamiento de las imágenes.

<su-usuario-en-dockerhub>/<nombre-del-repositorio>

Etiquete la imagen construida utilizando este formato:

```
$ DOCKERHUB_USERNAME=<su-usuario>
$ docker image tag docker-book-app ${DOCKERHUB_USERNAME}/docker-book-app
```

En este punto la imagen está lista para ser enviada a Docker Hub, pero primero es necesario configurar el acceso a esta plataforma desde el Terminal. La forma correcta de realizar el enlace es utilizando un *token* personal (*PAT, personal access token*) junto con el usuario. En este caso no debe utilizarse la contraseña del usuario. Para crear un token en Docker Hub acceda a las [configuraciones de seguridad de su usuario](#)<sup>107</sup> y genere un token con permisos de lectura y escritura.

Utilice el token generado para hacer login en Docker Hub.

```
$ export DOCKERHUB_PAT=<su-token>
$ echo $DOCKERHUB_PAT | docker login docker.io \
--username ${DOCKERHUB_USERNAME} \
--password-stdin
```

Login Succeeded

¡Perfecto! Todo listo para empujar la imagen hacia el registro.

```
$ docker image push ${DOCKERHUB_USERNAME}/docker-book-app
```

```
Using default tag: latest
The push refers to repository [docker.io/mmorejon/docker-book-app]
ceccbbaf318f: Pushed
latest: digest: sha256:932c6dc3152f892629d6124d2ca840b1e6b0e61959099332a60faa304d9191b6 s\
ize: 527
```

Compruebe la creación a través de la web o haciendo uso de la herramienta `hub-tool`.

```
$ hub-tool repo ls
```

REPOSITORY	DESCRIPTION	LAST UPDATE	PULLS	STARS	PRIVATE
mmorejon/docker-book-app		5 minutes ago	0	0	false

¡Enhorabuena! Ha creado su primer repositorio en Docker Hub, pero no va a ser el último :).

Le propongo como reto crear nuevamente el repositorio, pero esta vez a través de la web. Después de creado intente empujar la misma imagen. El resultado va a ser exactamente el mismo, con la diferencia que Docker Hub no tiene que crear el repositorio por usted.

Un detalle a tener en cuenta durante el reto son los nombres de los repositorios. No pueden existir dos repositorios con el mismo nombre en la plataforma, por lo tanto es necesario borrar el repositorio actual antes de crear uno nuevo a través de la interfaz web.

Para borrar el repositorio puede utilizar la web o la herramienta `hub-tool`.

---

<sup>107</sup><https://hub.docker.com/settings/security>

```
$ hub-tool repo rm ${DOCKERHUB_USERNAME}/docker-book-app
```

Con los conocimientos adquiridos en esta sección va poder almacenar las imágenes en Docker Hub, pero existe un detalle que necesita saber. Docker Hub permite solamente un repositorio privado, si desea tener más de uno tiene que cambiar de plan y pagar una suscripción. Sin embargo, existen registros de imágenes con ilimitados repositorios privados sin coste alguno. Para mostrar un ejemplo de ello se ha diseñado la siguiente sección, donde se utiliza el registro de imágenes de GitHub.

## Publicar imágenes en GitHub

GitHub permite almacenar el código fuente de sus aplicaciones, pero también brinda la posibilidad de almacenar otros artefactos digitales, como por ejemplo: imágenes de contenedores, gemas de ruby, paquetes npm, entre otros. Para conocer el listado completo de opciones puede hacer uso de [este enlace<sup>108</sup>](#).

De todas las opciones disponibles en GitHub, el almacenamiento de imágenes es el tema que interesa tratar en estos momentos. Entre las ventajas de utilizar el registro de imágenes está el hecho de poder tener todos los repositorios privados que deseé sin coste alguno. Los recursos en GitHub *no son infinitos*, [usted cuenta con 500MB<sup>109</sup>](#) de almacenamiento gratuito. Va a ser su responsabilidad crear imágenes bien pequeñas para optimizar su almacenamiento. En el capítulo anterior se explicaron algunas técnicas para lograr crear imágenes con poco peso.

Para publicar las imágenes en GitHub es necesario realizar los siguientes pasos:

- Tener creada una cuenta en GitHub.
- Crear un token con permisos de escritura de paquetes.
- Habilitar el acceso al registro en GitHub desde su terminal.
- Etiquetar la imagen con la estructura de nombre establecida por GitHub.
- Empujar la imagen desde su ordenador hacia GitHub.

Veamos a continuación algunos detalles a tener en cuenta a la hora de realizar estos pasos.

Seguramente usted tiene creada su cuenta en GitHub, pero si no es el caso, este puede ser el mejor momento para empezar. Utilice el siguiente enlace para acceder a la plataforma y crearse un usuario. Recuerde configurar el 2FA para fortalecer la seguridad de su cuenta.

<https://github.com/signup>

El segundo paso es crear un *token* con permisos de escrituras de paquetes. Este token va a ser su clave secreta para autenticarse con la plataforma. Para crear el token utilice el siguiente enlace: <https://github.com/settings/tokens/new>. Luego agregue una nota para recordar el propósito del token y

---

<sup>108</sup><https://docs.github.com/es/packages/learn-github-packages/introduction-to-github-packages>

<sup>109</sup><https://docs.github.com/es/billing/managing-billing-for-github-packages/about-billing-for-github-packages>

marque la casilla de escritura de paquetes. Va a notar que la sección *repo* se activa automáticamente quedando de la siguiente forma:

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

`write-packages`

What's this token for?

**Expiration \***

30 days The token will expire on Fri, Oct 14 2022

**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input checked="" type="checkbox"/> <b>repo:status</b>	Access commit status
<input checked="" type="checkbox"/> <b>repo_deployment</b>	Access deployment status
<input checked="" type="checkbox"/> <b>public_repo</b>	Access public repositories
<input checked="" type="checkbox"/> <b>repo:invite</b>	Access repository invitations
<input checked="" type="checkbox"/> <b>security_events</b>	Read and write security events
<input type="checkbox"/> <b>workflow</b>	Update GitHub Action workflows
<input checked="" type="checkbox"/> <b>write:packages</b>	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> <b>read:packages</b>	Download packages from GitHub Package Registry

Generar nuevo token en GitHub

Complete el proceso dando clic al final de la página en el botón *Generate Token*. Luego copie el token generado hacia un lugar seguro para utilizarlo en los próximos pasos.

El tercer paso es habilitar el acceso al registro en GitHub desde las líneas de comandos en Docker. Para resolver este punto ejecute el comando `docker login` haciendo uso de su usuario en GitHub y el token creado en el paso anterior. La documentación oficial para este proceso se encuentra en [este enlace<sup>110</sup>](#).

Adicione el valor del *token* y de su usuario en GitHub como variables de entorno en el terminal. En el comando de autenticación se hace referencia a `ghcr.io` porque es el registro donde se necesita acceso.

<sup>110</sup><https://docs.github.com/es/packages/working-with-a-github-packages-registry/working-with-the-container-registry>

```
$ GH_USERNAME=<su-usuario>
$ export CR_PAT=<su-token>
$ echo $CR_PAT | docker login ghcr.io \
  --username ${GH_USERNAME} \
  --password-stdin
```

Login Succeeded

El cuarto paso es etiquetar la imagen según la estructura de nombre definida por GitHub. La estructura establecida es la siguiente: `ghcr.io/OWNER/IMAGE_NAME`. Modifique los valores acorde a su usuario e imagen para etiquetar la imagen.

```
$ docker image tag docker-book-app ghcr.io/${GH_USERNAME}/docker-book-app
```

El último paso le va a resultar familiar, porque se ha utilizado en la sección anterior. Utilice el comando `image push` para empujar la imagen hacia el registro en GitHub.

```
$ docker image push ghcr.io/${GH_USERNAME}/docker-book-app

Using default tag: latest
The push refers to repository [ghcr.io/mmorejon/docker-book-app]
f209edd68017: Pushed
latest: digest: sha256:c4b7eeb65c8108e127be8484bfa7b2559f611791af327e3429b1c26ae01cd50c s\
ize: 527
```

Si ha recibido un mensaje similar a este significa que su imagen ha sido almacenada correctamente en GitHub. Acceda a la web de su usuario en GitHub, específicamente en la pestaña *Packages*, y podrá ver el paquete correspondiente a su imagen. Utilice el siguiente enlace modificando el `OWNER` por su usuario.

<https://github.com/OWNER?tab=packages>

Dedique unos minutos a explorar las opciones que brinda GitHub para esta imagen. Tenga en cuenta el siguiente detalle, la visibilidad de la imagen es privada por defecto. Esto significa que solamente usted puede acceder a ella. Si desea cambiar su configuración debe modificar su visibilidad a pública.

¡Enhorabuena! Ahora cuenta con otro registro de imágenes para compartir sus aplicaciones.

## Ejercicios para practicar lo aprendido

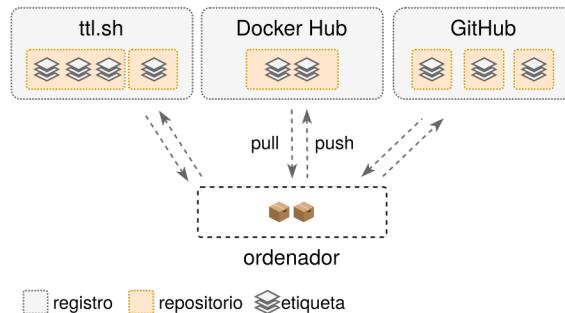
Después de haber incorporado nuevos conceptos y comandos es momento de practicar. Los ejercicios que aparecen a continuación tienen la intención de pasar un rato agradable delante del Terminal.

Tomando como referencia la plataforma cloud que utilice con mayor frecuencia identifique:

1. ¿Cuál es el nombre del Registro de imágenes?
2. ¿Cuál es la estructura de nombre definida para poder almacenar imágenes?
3. ¿Cuántas imágenes públicas y privadas puede almacenar?
4. ¿Cuál es el coste de almacenar imágenes en este registro?
5. Compare este registro de imágenes con las características mostradas en Docker Hub y GitHub.
6. Almacene la imagen docker-book-app en el registro de su plataforma. Seguramente va a necesitar adicionar una nueva etiqueta a la imagen.

## Resumen del capítulo

Durante el capítulo se ha explicado la importancia de utilizar *Registros* para almacenar las imágenes de los contenedores. Estos almacenes de imágenes permiten distribuir las aplicaciones de forma rápida y sencilla para luego desplegarlas en los servidores de producción.



Ejemplo de plataformas con registros

Cada *Registro* permite crear múltiples repositorios, donde cada repositorio almacena múltiples etiquetas de la imagen. Para almacenar las imágenes en los repositorios es requisito que cumplan con la estructura de nombre definida por el *Registro*. Para establecer los nombres en las imágenes se utilizan las etiquetas, las cuales pueden ser adicionadas o eliminadas según sea necesario.

También se mostraron tres ejemplos de registros: *ttl.sh*, Docker Hub y GitHub. En cada caso se explicaron los pasos a seguir para almacenar una imagen en cada uno de ellos. Existen muchos otros registros en la industria, cada uno con sus ventajas y desventajas, de aquí la necesidad que explore las opciones e identifique cuál es mejor para sus necesidades.

Después de tener la imagen distribuida en *Registros* todo queda listo para desplegar la aplicación en servidores de producción. Si le interesa saber cómo lograrlo, el próximo capítulo tiene las respuestas a sus preguntas.

# **Desplegar contenedores en la nube**

En desarrollo ...

# **Docker Compose V2**

Pendiente de desarrollo.

# **Recomendaciones y Próximos Pasos**

Pendiente de desarrollo.