

# Docker

**Primeros pasos  
y puesta en práctica  
de una arquitectura basada  
en micro-servicios**

Informática y Web



Jean-Philippe  
**GOUIGOUX**



**ε**  
Colección

**epsilon**

Archivos complementarios  
para descarga



# Docker

## Primeros pasos y puesta en práctica de una arquitectura basada en micro-servicios

Este libro sobre **Docker** se dirige a todos los **desarrolladores, arquitectos y administradores**, que deseen adquirir un **conocimiento estructurado sobre Docker**, basándose en la realización práctica de un ejemplo específico de despliegue de una aplicación, basada en contenedores. Tener un conocimiento mínimo de Linux y de sus redes TCP/IP, es un requisito previo imprescindible para sacar el máximo beneficio posible de este libro, que se organiza en dos partes diferentes.

La primera de ellas detalla los **mecanismos que forman los fundamentos de Docker**, con ejemplos de operaciones basadas en **ejercicios prácticos**. Docker forma parte de un ecosistema muy poblado y extremadamente activo. El autor se centra en proporcionar al lector **bases sólidas sobre su funcionamiento** y sobre sus conceptos más importantes, para que su uso sea eficaz y para cubrir las necesidades básicas de los usuarios profesionales (**creación de imágenes sencillas y sólidas, conocimiento sobre las buenas prácticas, seguridad, etc.**).

En una segunda parte, el autor implementa una **infraestructura completa** alrededor de una aplicación de ejemplo, basada en una **arquitectura de micro-servicios**. El punto de vista adoptado por el autor es el de un fabricante de software que necesita implantar servicios de software en una **arquitectura flexible y evolutiva**. Esta puesta en práctica sobre una aplicación realista, permite describir los **trucos de despliegue** de contenedores para Java, Core .NET, Python y Node.js/AngularJS. La aplicación que se crea de esta manera, se **despliega** posteriormente **en un cluster** de máquinas gestionadas por Docker Swarm.

Hay elementos adicionales disponibles para su descarga en esta página. Así, el lector puede implementar todos los ejemplos que se tratan en el libro.

### Los capítulos del libro:

Prólogo – Introducción a Docker – Primeros pasos – Creación de sus propias imágenes – Instalación de un registro privado –

Implementación de una arquitectura de software – Despliegue en un cluster – Ir aún más allá con Docker

---

Jean-Philippe GOUIGOUX

**Jean-Philippe GOUIGOUX** es director técnico de un fabricante de software, para el que ha rediseñado la arquitectura alrededor de micro-servicios urbanizados. Está reconocido como Microsoft MVP (Most Valuable Professional) en varias especialidades desde hace 8 años. Es autor de varios libros y artículos científicos, interviene de manera regular en la universidad, así como en conferencias de Microsoft Experiences, Agile Tour o BreizhCamp, para compartir sus conocimientos. Este libro sobre Docker es su ocasión para ayudar a los lectores a adoptar una nueva herramienta de manera estructurada, que está en camino de revolucionar el mundo de la informática, como hizo en su época la virtualización.

## Introducción

No pasa una semana sin un anuncio Docker: una innovación sobre su uso en arquitecturas orientadas a servicios, una nueva herramienta en el ecosistema, un producto comercial u open source para solucionar una limitación, etc. Muchos de estos anuncios rápidamente pierden interés. Aparecen algunas buenas prácticas, herramientas que progresivamente se integran en Docker pero al final, muchas de ellas ni siquiera pasan al estado de versión beta.

Este hecho, ya presente en la primera edición del presente libro, siempre es cierto: el ecosistema alrededor de Docker se ha desarrollado mucho en los últimos dos años, las tecnologías que lo conforman ahora forman parte del producto y otras han desaparecido. Por el contrario, las limitaciones de Docker (utilización en red, problemas de seguridad, etc), son historia, incluso si el dominio de los contenedores continúa evolucionado tanto, hay cierta estabilidad en las funciones básicas.

El objetivo del presente libro es permitirle ordenar sus ideas, utilizando un enfoque centrado alrededor de las funcionalidades principales de Docker y priorizando la comprensión profunda de lo que la herramienta permite, más que intentar cubrir todos los aspectos alrededor de Docker. La supervisión, seguridad y el resto de aspectos para los que las buenas prácticas todavía no existen, se dejan voluntariamente a un lado.

Se abordará la Docker Machine y el modo Swarm de Docker, porque se tratan de módulos estables, pero sin entrar en detalles, que es el objeto central de un segundo libro publicado en Ediciones ENI, llamado "Despliegue de micro-servicios en Linux o Windows (Docker Swarm, Docker Compose, Docker Machine)".

Por otro lado, la comprensión de los mecanismos que forman los fundamentos de Docker, será objeto de atención particular. Para ello, hay un segundo capítulo dedicado a las implantaciones sencillas pero prácticas. Siempre con el objetivo de ir a lo fundamental, el libro no se detendrá en todas las opciones de los comandos de Docker, sino que se estudiará de manera muy detallada el funcionamiento concreto y los efectos de los más importantes.

Se dedica un capítulo entero a la creación de imágenes Docker porque, más allá de usos innovadores, montajes exóticos y otras curiosidades de utilización de Docker (en Internet abundan),

actualmente la gran mayoría de las necesidades de los usuarios profesionales se centran en crear imágenes sencillas, pero de calidad. El siguiente capítulo está más dedicado a los profesionales, porque presenta la implantación de un registro privado. Cuando se compilan las imágenes, es necesario ponerlas a disposición de los consumidores y además, mientras que el registro público de Docker no sea una solución aceptable, es necesaria la implantación de un registro privado. Sin embargo, esta última se ha hecho más sencilla que durante la primera edición del presente libro, con numerosas soluciones industriales en línea, que hacen que el despliegue manual de un registro ya no sea útil.

El objetivo del libro es permitir que un lector con algunos conocimientos de Linux y redes TCP/IP, consiga un conocimiento estructurado de Docker. No es necesario un conocimiento amplio pero sí una base, a partir de la que el lector se sentirá preparado para abordar cualquier documentación relativa a un aspecto avanzado, armado con una correcta comprensión de los conceptos básicos. Por esto, la segunda parte del libro se articula alrededor de una aplicación de ejemplo, de manera que se facilita la mirada crítica sobre una puesta en práctica real de los conceptos aprendidos en la primera parte. Aunque este aspecto no se estudia en profundidad, aprovechando la ocasión, se mostrará el despliegue de una aplicación en un cluster Docker.

La aplicación que servirá de ejemplo solo se puede considerar como una sencilla prueba de concepto de una arquitectura de micro-servicios, que utilizan contenedores. Se compone de seis módulos independientes y, aunque no implemente una funcionalidad realista, su arquitectura es parecida a las que existen en el mercado para las arquitecturas de micro-servicios. Normalmente, Docker se utiliza como una pieza fundamental de estas arquitecturas, porque permite realizar las separaciones de código que implican un desacoplamiento funcional, limitando el sobre coste en términos de recursos. Para que cada lector encuentre su camino respecto a su lenguaje preferido, casi todos los módulos de software utilizan una tecnología diferente. Esto nos permitirá una gama completa de consejos y trucos necesarios para un uso práctico de Docker.

Después de haber explicado paso a paso cómo montar esta arquitectura, el libro abordará el problema del despliegue siguiendo un primer punto de vista sobre la automatización y un segundo, sobre la puesta en disposición en un cluster de máquinas Docker. Para terminar, un último capítulo detallará los modos de utilización de Docker en una fábrica de software y mostrará algunas perspectivas del futuro de esta sorprendente herramienta.


Docker y las tecnologías de contenedores ya han cambiado profundamente el mundo del despliegue informático, de la misma manera que la virtualización lo hizo hace veinte años. Pero por el momento, el presente libro comienza por los conceptos básicos y se plantea el objetivo de ofrecerle una comprensión profunda de lo que forma, oculto bajo las herramientas y las líneas de comandos, el núcleo de Docker y su potencial revolucionario.

# 1. INTRODUCCIÓN A DOCKER

## Docker como alternativa ligera a la virtualización

Durante la primera edición del presente libro, Docker era una herramienta todavía joven, alrededor de la que se podía percibir una pasión real. Como sucede habitualmente en estos casos, existía cierta confusión sobre el alcance exacto de la tecnología, y el libro tenía como objetivo no comenzar directamente con una explicación de las funcionalidades de Docker, sino concretar la intención de la herramienta, es decir, el papel que se proponía jugar en un SI (Sistema Informático).

Unos dos años más tarde, y aunque Docker todavía era una herramienta importante pero casi estándar entre las posibilidades que se ofrecían para desplegar aplicaciones, esta necesidad de entender el por qué de la tecnología antes de adentrarse en el cómo, sigue estando de actualidad. Docker se interesa por todo; se beneficia de un rico ecosistema, ha sido adoptado por todos los proveedores de cloud y dispone de una implementación Windows además de su origen Linux, pero volver a los fundamentos y comprender por qué nació Docker, permite entender mejor su funcionamiento.

 Los capítulos que siguen usarán únicamente Docker en su versión Linux. Aunque existe una versión para Windows 10 y Windows Server 2016, los comandos y conceptos mostrados en el presente libro son exactamente los mismos, por lo que nos quedamos con la versión original para este libro sobre los fundamentos.

## 1. El enfoque por virtualización

En una visión global, Docker idealmente se describe como un conjunto de herramientas que facilitan la implantación de despliegues informáticos, como si fuera una virtualización ligera, aunque no sea propiamente dicho una técnica de virtualización. Con el objetivo de precisar en qué consiste esta ligereza, a continuación se muestra un primer esquema de una arquitectura

virtual tradicional, tal y como la encontraríamos en herramientas como VMware, VirtualBox o Hyper-V.



El tamaño de los bloques simboliza el espacio ocupado por los diferentes componentes, pero las proporciones no muestran completamente la enorme cantidad de espacio que necesita un sistema operativo (OS, para *Operating System*). En ese caso, el esquema sería legible ya que los OS modernos incorporan capas de indirección, funcionalidades diversas y componentes únicamente presentes por razones de compatibilidad, incluso de redundancia, instalando por ejemplo los componentes multimedia en un servidor web, que nunca necesitará utilizarlos.

El hecho de que una máquina virtual (VM) necesite la re-instalación de un OS completo, más allá de la máquina física, hace el problema particularmente significativo. Además, es necesario añadir que en la práctica industrial, la virtualización raramente sirve para desarrollar OS diferentes en un mismo soporte físico. Las funcionalidades que se buscan son principalmente la facilidad en el despliegue de una nueva máquina, duplicando una imagen ya existente, así como la capacidad de crear una estanqueidad fuerte entre dos entornos. Los sistemas operativos son idénticos: la redundancia, ya sea en memoria RAM o en los discos, es particularmente marcada y por tanto, estos recursos generalmente se desperdician.

## 2. Docker para optimizar el uso de los recursos

Existen enfoques para poner en común fragmentos de imágenes de VM, pero Vagrant, por ejemplo, simplemente va a compartir el espacio en disco para las máquinas basadas en la misma imagen. Durante el arranque, la VM necesitará su propio espacio en memoria RAM.

De la misma manera, los hipervisores recientes son capaces de hacer una sobre-asignación de recursos en memoria, de manera que se pueda instalar más VM que la RAM (*Random Access Memory*, es decir la memoria RAM) que teóricamente puede soportar, jugando con el hecho de que todas las máquinas no la solicitarán necesariamente al mismo momento. Pero cuando hay dos máquinas funcionando, hay funcionalidades repetidas y a menudo, muchas de ellas no se utilizan (incluso nunca). El precio



a pagar en términos de rendimiento y de dificultad de equilibrar las cargas, es muy elevado.

Lo ideal, cuando el OS o grandes porciones de él se comparten, sería que las aplicaciones que deben permanecer estancas puedan acceder a las funcionalidades comunes (OS principalmente, así como las librerías), conservando una garantía de estanqueidad. Esto es lo que permite la tecnología de contenedores, donde la herramienta Docker facilita la explotación. El siguiente esquema muestra una versión (muy simplificada, los detalles y casos particulares se estudiarán más adelante), de la manera en la que Docker permite ahorrar memoria.

images/esquema2.png

Evidentemente, el espacio ganado comprimiendo todo aquello que es necesario para que las aplicaciones funcionen, permite (lo que, por otro parte es el objetivo) permitir que funcionen más aplicaciones con la misma cantidad de recursos físicos.

Por tanto, quizás el esquema correcto sea el siguiente.

images/esquema3.png

De nuevo, por razones de expresividad, las proporciones visuales de los esquemas no tienen nada que ver con la realidad. En la práctica, el ratio entre el número de aplicaciones desplegadas en los contenedores y el número de aplicaciones ejecutadas en modo virtual, será superior al ratio de cinco a dos, ilustrado en los diagramas anteriores. El ratio efectivo depende de muchos factores (tipos de OS homogéneos o no, dependencias comunes entre las aplicaciones, etc.) pero el retorno de experiencia hace que el número vaya de 5 a 80. Por supuesto, este último valor corresponde a un caso particular de adaptación, y no debe implicar centrarse en este criterio. Lo principal se centra en la mejora de uso.

Por supuesto, esta fuerte reducción de la redundancia es muy positiva, desde el punto de vista ecológico y económico. Las ganancias de tipo GreenIT (informática "verde") en las arquitecturas de tipo cluster o cloud son evidentes, y tienen que ver con la velocidad sorprendente de adopción de Docker. Pero es necesario reconocer que para el administrador de aplicaciones, la ecología no es normalmente un criterio fundamental, mientras que la velocidad de implantación de un contenedor gana adeptos. En efecto, un contenedor arranca en varios centenares de milisegundos la mayor parte de las veces, mientras que una

máquina virtual llevará varias decenas de segundos en el mejor de los casos. Estos segundos, así como el uso de menos recursos, han proporcionado muchos usos hasta ahora impensables o al menos, difícilmente aplicables en la práctica.

Para dar solo un ejemplo, veamos el caso de un desarrollador que desea ejecutar pruebas unitarias en su código. Una buena práctica es cada vez partir de un entorno virgen. Las máquinas virtuales son ideales para esto, porque se pueden crear una sola vez y después, cada vez que se detiene solicitar eliminar los cambios realizados desde el inicio. Pero, en un enfoque de tipo TDD (*Test-Driven Development*), es decir desarrollo controlado por pruebas, se supone que el desarrollador vuelve a ejecutar todas las pruebas cada vez que se modifica o añade una funcionalidad en su código, lo que significa que potencialmente debe hacerlo decenas de veces cada hora. Cuando las pruebas unitarias alcanzan una duración de ejecución que sobrepasa el minuto, este bucle de iteración se hace más lento. El confort aportado con un inicio en menos de un segundo respecto a 30 o más, hace que realmente el TDD se respete, abriendo la vía a sustanciales mejoras en la calidad del código.

### 3. Posicionamiento de Docker respecto a la virtualización

Sobre todo, Docker realiza esta economía de recursos sin sacrificar la estanqueidad de los contenedores, que se garantiza a nivel del OS subyacente. En una sección más delante de este capítulo, se volverá en más en detalle sobre los métodos utilizados para esto. Por supuesto, como en todo proceso de software, existen defectos y conviene dar seguridad a esta estanqueidad, utilizando medios adicionales y hacer un seguimiento escrupuloso de las acciones correctivas. Pero se puede mantener el mismo discurso para los hipervisores tradicionales.

Como hemos podido ver en los esquemas anteriores, el enfoque Docker permite abstraerse de cualquier hipervisor. En su lugar se sustituye por esto que simplemente hemos llamado "Docker", sin precisar por el momento qué incluye esta pieza. Por el momento, simplemente indicamos que no se trata ni de un emulador ni de un motor de virtualización, ni de un hipervisor: Docker es un software cliente-servidor basado en funcionalidades de bajo nivel del nodo Linux (y ahora también Windows, como se ha explicado más atrás). En este sentido no podemos hablar de virtualización ligera para Docker, incluso si funcionalmente una gran parte de los usos se corresponden con esta manera de simplificar el producto.

Para terminar esta sección, puede ser instructivo detenerse en las diferencias y similitudes de los dos enfoques. Como se ha dicho anteriormente, ninguno es infalible desde el punto de vista de la seguridad de la estanqueidad del contenido. Se han encontrado fallos en Docker, como en la mayor parte de los hipervisores, que permiten acceder a los datos de otra VM u otro contenedor, incluso al sistema host en sí mismo. De la misma manera, las funcionalidades básicas ofrecidas por una u otra tecnología, son muy parecidas: creación y ejecución de una entidad, parada/pausa/reinicio, asociación a los recursos físicos como la red o el sistema de archivos, etc.

Los puntos de divergencia principales son que el enfoque de virtualización consiste en simular el funcionamiento de una máquina física, mientras que Docker y las tecnologías de contenedores en general, parten del principio de que la máquina es central, y que la estanqueidad debe estar garantizada por las funcionalidades del OS en sí mismo. Por lo tanto, el enfoque virtualizado es más estricto, porque a priori se encarga de la estanqueidad, de ahí la extendida idea de una mayor garantía desde este punto de vista.

Otra importante diferencia reside en la riqueza de los ecosistemas. Las soluciones de virtualización se benefician de veinte años de experiencia, con una gran difusión. Se han añadido y estabilizado numerosas herramientas adicionales (migración de máquinas sobre la marcha, provisioning, gestión virtual de la red, etc.). Por su parte, el enfoque basado en contenedores no es nuevo. Existe desde hace mucho tiempo en los mainframes y ha experimentado un repunte en su popularidad (aunque limitado), durante la implantación en los sistemas operativos más modernos. Pero su utilización, limitado a un círculo de iniciados, hace que las herramientas no abundan.

En el caso particular de Docker, que recientemente se ha convertido en una herramienta con gran difusión y muy estable, se ha implementado un ecosistema alrededor de la tecnología, con las aportaciones resultado de la comunidad, que se han incorporado en el software de control (como Docker Compose, resultado del software Fig), así como numerosas evoluciones realizadas por Docker Inc., la empresa que garantiza la parte principal del desarrollo de Docker. Todos estos módulos adicionales hacen que Docker pueda aspirar a una exhaustividad, incluso madurez, parecida a la de las tecnologías de virtualización. En particular, la gestión de la red ha sufrido estos dos últimos años una transformación que le ha permitido alcanzar un excelente nivel, la tecnología Swarm integrada en el producto, sustituye favorablemente todos los intentos de mejora de aquello que

representaba un punto de sufrimiento hace apenas dos años. De la misma manera, todos los problemas de juventud desde el punto de vista de la seguridad, se han ido limando y, aunque un producto tan reciente no puede aspirar a la seguridad de funcionamiento de hipervisores en producción después de una quincena de años o más, sin lugar a dudas Docker es apto para la producción en los entornos más sensibles, donde las arquitecturas compartimentadas que permite, son incluso un activo para la seguridad en profundidad de los sistemas (principio de *Defense in Depth*, o defensa en profundidad).


Al final, hoy en día Docker complementa la virtualización, más que sustituirla. La virtualización conserva claramente algunos aspectos en los que Docker no pretende aventurarse, sin correr el riesgo de distorsionarse al querer cubrir demasiados dominios diferentes. Pero el enfoque por contenedor ha ganado mucha popularidad después de haber visto durante mucho tiempo, su superioridad técnica reducida a nichos para expertos y ahora, sin embargo, representa el método preferido de despliegue para las arquitecturas sofisticadas. En pocas palabras, Docker ha realizado plenamente su papel de facilitador en la utilización de esta tecnología de contenedores.

## Principio de los contenedores

El aporte principal de Docker consiste, como se ha explicado en la sección anterior, en una reducción drástica de los recursos utilizados, conservando una estanqueidad entre las aplicaciones. Pero, una vez más, las tecnologías de compartimentación de memoria existían antes de Docker. Este último ha añadido un determinado número de funcionalidades adicionales, que han permitido su adopción más amplia.

En el primer grupo de estas funcionalidades, se encuentra la normalización de los contenedores: Docker ha tenido éxito allí donde los enfoques puramente técnicos se han detenido, porque ha hecho extremadamente sencilla la recuperación de contenedores existentes y su creación, usando métodos estándares. En la historia de la informática, la normalización y la estandarización, casi siempre han sido requisitos previos para la difusión amplia de una tecnología dada.

El API Docker se ha estandarizado por medio de dos especificaciones de Open Container Initiative, un consorcio web abierto de normalización de la gestión de contenedores al que se adhiere Docker, así como numerosas grandes empresas de informática. La primera de estas especificaciones afecta a la gestión de la ejecución de los contenedores (y por tanto al API para arrancarlas, detenerlas, etc.) y la segunda, la definición de las imágenes. Hay más información en <https://www.opencontainers.org/>.

 Un API (de Application Programming Interface), es un conjunto de funciones que permiten controlar una funcionalidad informática por medio de otro programa, al que llamamos "cliente".

Es notable que Microsoft, para su implementación de la gestión de contenedores en Windows, haya seguido escrupulosamente este enfoque normalizado y ha ofrecido un API 100 % compatible con el de Docker en Linux lo que permite, por ejemplo, controlar un servidor Docker en Linux con un cliente Docker para Windows. De la misma manera, es este respeto de las normas lo que permite realizar todos los ejemplos del presente libro en Linux sin que esto suponga el menor problema para un usuario de Windows. Para transponer estos ejemplos no es necesaria ninguna modificación de

comandos, cambio de nombres de argumentos u otra acción de transposición.

## 1. Principio de los contenedores industriales

El nombre de contenedor es revelador, en el sentido que permite hacer un símil con los contenedores industriales utilizados para el transporte de carga. Hasta los años 1950, la mayor parte de las veces los contenedores para la carga se hacían por encargo, según la forma, el volumen y la masa del contenido, así como las capacidades de realización de un contenedor por parte del remitente. Como resultado los tamaños dispares de los paquetes, aumentaba la dificultad para optimizar los cargamentos.

En 1956, Malcom McLean, un contratista americano en transportes, desarrolló un sistema sencillo de transferencia de cajas entre barcos y camiones, evitando de esta manera transbordar las mercancías una a una hasta el remolque. El éxito de esta prueba se propagó y en 1967, el ISO (*International Organization for Standardization*) normaliza tres tamaños estándares de contenedores para el transporte. A pesar del alcance mundial de las modificaciones necesarias, el transporte marítimo y por camión a nivel global se transforma rápidamente, gracias a la estandarización de los contenedores. Esto permite optimizar prácticamente toda la cadena de transporte:

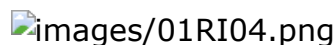
- Facilidad de la carga, porque cualquiera pueda hacerse con un contenedor y cargarlo a su ritmo y después, simplemente contactar con un transportista que aceptará su cargamento.
- Mejora en el mantenimiento del contenedor: el tamaño era estándar, las abrazaderas de carga ya no tienen que apretar la caja (o peor, tener que cambiarse para cada cargamento diferente), sino que simplemente se pueden utilizar los enganches que encajan en las esquinas reforzadas de los contenedores. La duración de vida de estos últimos se alarga considerablemente.
- Aumento de la velocidad de transporte: todas las mercancías contenidas en un bloque se mueven al mismo tiempo, en lugar de trabajar paquete a paquete.
- Almacenamiento optimizado: las dimensiones normalizadas permiten apilar y ajustar los espacios de almacenamiento sin perder espacio. Los barcos porta-contenedores se dimensionan en función de los contenedores (y no a la inversa, sinónimo de pérdida de espacio).

- Capacidad de intercambio en todas las partes del mundo: un contenedor dañado se puede reparar fácilmente o sustituir sin tener que devolverlo a su remitente.

## 2. Docker y el enfoque normalizado

El producto Docker apareció en el paisaje informático a comienzos del año 2013, cuando dotCloud, una empresa proveedora de PaaS (*Platform as a Service*) publicó en modo open source, un conjunto de herramientas de las que era propietaria.

El nombre de Docker, así como su logo (una ballena transportando contenedores), anunciaba claramente el objetivo que tenía la empresa con el producto, a partir de ese momento llamada Docker Incorporated. A pesar de que el producto era demasiado joven para que se pudiera comprobar su aportación a la estandarización de los contenedores industriales para el transporte, a día de hoy Docker está muy bien situado para convertirse en los próximos años en el modo habitual de despliegue de aplicaciones de software.



Para seguir con esta comparación, a continuación se muestran los puntos fuertes que Docker puede presentar para justificar esta ambición:

- Facilidad de carga de un proceso en un contenedor: un contenedor vacío se puede recuperar e instanciar en una línea de comandos (el software Docker y un ordenador conectado a Internet son los únicos requisitos previos) y la ejecución de un proceso en el contenedor, se hace exactamente de la misma manera que en una máquina local.
- Esta carga se pueda automatizar a través de un archivo que contenga un texto escrito en un formato que describa las etapas de la carga del proceso, en el contenedor.
- Mantenimiento del contenedor: cualquier persona que haya creado un contenedor rellenándolo con un proceso configurado como quiera puede difundirlo por Internet, teniendo la garantía de que cualquier otra persona podrá ejecutar el contenedor.
- Rapidez de ejecución: el contenedor agrupa toda la configuración, los entregables, los recursos y todo lo que es útil para el programa soportado, su ejecución se hace casi de

manera inmediata, y sin que sea necesario tiempo de instalación.

- Capacidad de intercambio: cualquier persona que ejecuta el contenedor obtendrá exactamente el mismo resultado que el emisor. Este enfoque estándar permite garantizar una misma ejecución de los programas, sea cual sea el sistema subyacente, siempre que se proporcione el programa Docker.
- Independencia respecto a la infraestructura subyacente que tengan los contenedores: estos últimos se pueden ejecutar de manera indiferente en un servidor Docker local a una máquina, en un cluster de servidores Docker instalados en las máquinas en red (hablamos del modo Swarm de Docker, sobre el que volveremos más adelante), en un cloud que ofrezca la funcionalidad de Container as a Service, etc.



## Los aspectos principales de Docker

Como se ha explicado más atrás, Docker se basa en los fundamentos existentes y alcanza un determinado éxito porque convierte en sencillo el uso de estas tecnologías subyacentes. Sin entrar en detalles (Docker tiene por objetivo justamente ocultar este mecanismo complejo), siempre conviene echar un vistazo más profundo para entender mejor el funcionamiento de la máquina Docker.

### 1. Espacio de nombres

En la base de la tecnología de contenedores, se encuentra la noción de espacio de nombres. Estos "espacios de nomenclatura" permiten crear conjuntos de recursos gestionados por Linux, mientras que de manera tradicional se gestionaban en un único bloque. Por ejemplo, los procesos se identifican por un número, accesible en una lista de todos los procesos ejecutados por el sistema Linux. Se trata del identificador de proceso o PID.

Fuera de los espacios de nombres, el comando `ps` envía la lista completa de los procesos activos en la máquina Linux. El principio del espacio de nombres es realizar las visualizaciones restringidas de estos recursos, según otra lógica. Por ejemplo, en el espacio de nombres 1 solo es visible el proceso de PID 10, y aparece bajo un valor de PID local de 3. En un segundo espacio de nombres, son visibles los procesos 11 y 12 pero con los identificadores 3 y 4.



De esta manera, allí donde el nodo Linux impide tener el mismo identificador para dos procesos separados es posible, con la condición de estar en dos espacios de nombres diferentes, tener dos veces el mismo PID. De una determinada manera, se puede decir que el identificador del proceso es, espacio de nombres1.pid3. En este caso, los espacios de nombres funcionan un poco como los espacios de nomenclatura de las clases en los lenguajes orientados a objetos, como Java o C#.

Además, cuando el usuario del sistema se localiza en un espacio de nombres dado, utilizando el comando `ps` solo ve los procesos

asociados a este espacio de nombres. Está aislado del resto de procesos gestionados por el nodo Linux.

Esta es la capacidad que se corresponde con prefijar los identificadores que se implementa por los espacios de nombres en el nodo Linux. El ejemplo dado para los procesos, se puede aplicar al resto de recursos que soporten la partición por esta tecnología. En efecto, existen espacios de nomenclatura para los procesos (`pid`), como se ha explicado anteriormente, así como para las interfaces de red (`net`), las comunicaciones inter-proceso (`ipc`), los puntos de montaje (`mnt`), etc.

## 2. Cgroups


Más allá de la posibilidad que ofrecen los espacios de nombres para tener una visión parcial de las funcionalidades ofrecidas por el nodo, también es necesario poder particionar el acceso a los recursos, como la memoria, los accesos a disco o las franjas de CPU (*Central Processing Unit*, es decir el microprocesador). El hecho de que la aplicación en un espacio de nombres solo vea los recursos de su espacio de nombres es una cosa, pero también es necesario garantizar que no pueda tener recursos de fuera. Este es el papel de los controles de grupos o cgroups.

Evidentemente, los cgroups funcionan en estrecha colaboración con los espacios de nombres, aunque se trate técnicamente de dos funcionalidades distintas.

Por ejemplo, los cgroups permiten asignar únicamente el 15% de la CPU disponible a un proceso, incluso a un conjunto de procesos (típicamente los procesos relacionados con un sistema de virtualización nivel OS).

## 3. LXC

Las dos tecnologías anteriores eran relativamente complejas de manipular utilizando la línea de comandos. LXC nace como un primer enfoque orientado a simplificar su uso. El paradigma utilizado es el de los contenedores, de ahí el nombre LXC (*Linux Containers*). La noción de contenedor agrupa las asignaciones de recursos y el aislamiento por medio de espacio de nombres/cgroups, de manera que se facilite la implantación de conjuntos de proceso aislados, los unos de los otros.

 Observe que el principio de aislamiento por contenedores existía ya en otros OS, como FreeBSD (tecnología Jails) o Solaris (donde se llamaban zonas). El principio maestro detrás de los contenedores es ofrecer un aislamiento a nivel del sistema: es el sistema operativo el que se encarga de la estanqueidad de los recursos. Por supuesto, este enfoque no es absoluto y depende de la programación correcta de los mecanismos. Pero el sobre coste de una estanqueidad de nivel superior, emulando completamente una máquina, es enorme y los hipervisores no están libres de fallos, aunque normalmente sean de otro tipo. Cuando sea necesario tener una garantía de estanqueidad de muy alto nivel (operación de datos altamente confidenciales, secretos militares, etc.), el coste limitado de los recursos de tipo hardware, normalmente hace concluir que la mejor garantía es una separación física de las máquinas

LXC no ha tenido el éxito que la tecnología merece, en parte debido a una falta de documentación y por otro lado, debido a un soporte quizás degradado en los OS Linux diferentes de Ubuntu. Docker ha tomado el desafío de poner a disposición en el mayor número posible, basándose para ello inicialmente en LXC.

## 4. Libcontainer

Desde la versión 0.9, Docker ha eliminado su dependencia con LXC, basándose en una interfaz más contractual, que define de manera estándar las API que gestionan el acceso a los recursos. Esto permite abrir la puerta a otros OS (más adelante volveremos sobre el caso concreto de Windows). Esta interfaz se conoce con el nombre de libcontainer y Docker ha realizado la primera implementación apuntando al nodo Linux.

La API libcontainer permite utilizar los espacios de nombres sin tener que depender de una implementación particular. Prepara Docker para una utilización en OS diferentes a Linux.

 Images/esquema5.png

Por supuesto, sigue siendo posible utilizar Docker con LXC, tal y como se representa en la parte derecha del diagrama anterior. Por el contrario, la limpieza estructural aportada por el hecho de no depender de un módulo, sino únicamente de una definición estándar de API (parte izquierda), hace pensar que el código de acceso a LXC desaparecerá eventualmente.

La implementación de libcontainer en el nodo Linux, creado por Docker (la empresa), se realiza en lenguaje Go, así como muchas otras partes de Docker (el producto).

Es notable que es esta capa de indirección la que se ha utilizado para realizar una implementación Windows de Docker, empezando por una implementación de libcontainer que ya no se basa en LXC, sino en las tecnologías correspondientes en Windows, a saber, Server Silos en lugar de los espacios de nombres y Windows Job Objects, que sustituye a los cgroups de Linux. La tecnología Microsoft Drawbridge se había diseñado como la base de la versión Windows de Docker, pero al final resulta que no estaba adaptada.

## 5. Sistema de archivos en capas

### a. Principio de aislamiento de los archivos

Hasta ahora, se ha hablado mucho sobre el aislamiento de los recursos, pero Docker también se basa en otra tecnología muy importante de Linux, a saber la gestión de archivos por capas, que ofrecen algunos sistemas de archivos.

De cierta manera, este enfoque prolonga el aislamiento situándolo a nivel de cada archivo, pero vamos a ver que además de la estanqueidad, este enfoque por capas permite la reutilización y por tanto, una importante economía de recursos.

### b. Enfoque por virtualización

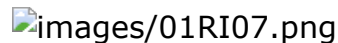
El ejemplo para ilustrar este enfoque, será el de una aplicación tradicional compuesta del tipo 3-tiers:

- una capa visual en forma de interfaz web,
- un servidor que expone los servicios web,
- una base de datos.

Por razones operativas, es muy habitual no multiplicar demasiado las tecnologías soportadas en los sistemas de software. De la misma manera, es lógico imaginar que los 3-tiers se basen en la misma versión de Linux (por ejemplo, una CentOS 7) y que la capa visual y los servicios utilicen el mismo servidor web (por ejemplo, un Apache Web Server 2.4.12). En primer lugar, el aspecto específico de cada servidor consiste en una aplicación web gráfica, seguido por una aplicación web que expone los

servicios en forma de API y, en último lugar, una base de datos (PostgreSQL 9 en nuestro ejemplo).

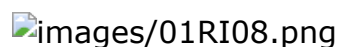
En un enfoque virtualizado, esto se traduce por el siguiente montaje, simplificado voluntariamente para utilizar un solo servidor por tiers (abordaremos la redundancia y el equilibrio de carga en su momento).



El desperdicio de recursos a nivel de los archivos, se debe comparar con el provocado por la repetición del consumo de recursos de sistema por tres máquinas virtuales, cada una con su sistema operativo. Anteriormente hemos visto que el enfoque de Docker consistía en garantizar la estanqueidad a nivel de los recursos del OS y por lo tanto, compartir mejor la CPU, la memoria RAM, los accesos a la red, la gestión de los procesos, el ancho de banda hacia los discos, etc. Se observa que Docker también permite ahorrar a nivel de los archivos, compartiendo aquello que es común, permitiendo conservar una gran estanqueidad.

### **c. Herramienta de los sistemas de archivos en capas**

En la práctica, los contenedores utilizan imágenes (se trata del término realmente utilizado) con contenidos de archivos y estos, se agrupan en capas que se añaden. De esta manera, es posible realizar una puesta en común de los archivos, como se muestra en el siguiente diagrama.



En lugar de duplicar los archivos del sistema, así como del servidor web en dos o tres máquinas virtuales, las capas que componen las imágenes de los contenedores hacen referencia las unas a las otras. Cuando se inicia un contenedor de tipo frontal web, se basa en una imagen Web GUI (*Graphical User Interface*) que hace referencia a la imagen Apache, que a su vez se basa en la imagen CentOS. La puesta en común de los niveles intermedios, permite obtener imágenes muy ligeras, siendo las características específicas de cada una respecto a la parte que permanece similar muy bajas.

En particular, la diferencia entre el front web y el tiers de servicios web, se limita a algunos archivos que forman el sitio web y al API. Por ejemplo, la imagen web GUI contiene:

- archivos `.html`,
- scripts JavaScript,
- hojas de estilo `.css`.

mientras que la imagen Web API contiene, si estamos en una pila Mono:

- El módulo Apache `mod-mono`, para que funcione Mono (la implementación de .NET en Linux) en el servidor web,
- archivos `.asmx`,
- librerías `.dll`,
- un archivo `web.config` para configurar la aplicación.

La imagen común Apache, de la que dependen las dos anteriores, contiene:

- el archivo de configuración del servidor web `/etc/apache2/apache2.conf`,
- los archivos binarios en `/usr/sbin`,
- la documentación en `/usr/share/doc`,
- etc.

La instanciación de un contenedor fusiona de manera dinámica todas estas capas y presenta un sistema de archivos que contiene todo lo que es necesario:



El ahorro en el tamaño de las imágenes es particularmente considerable en el caso de su difusión por Internet, donde en lugar de transmitir los archivos correspondientes a los discos virtuales de varias decenas de gigabytes (lo que siempre ha sido una de las limitaciones para la utilización de la virtualización para las pruebas y la depuración de las aplicaciones), ahora se pueden enviar archivos de algunas decenas de megabytes únicamente.

#### **d. Gestión de las modificaciones de archivos**

Hasta ahora, el sistema de archivos por capas no parecía nada especial realmente: solo agrupa los archivos que provienen de varias capas, para crear una única imagen que los presenta todos durante la instanciación de un contenedor. Un caso particular de

operación va a mostrar toda la potencia del concepto, es decir, la gestión de un archivo presente en varias capas.

En nuestro ejemplo, nos podemos imaginar que las dos aplicaciones necesitan argumentos diferentes para el servidor Apache subyacente. Por lo tanto, el archivo `/etc/apache2/apache2.conf` debe ser diferente en las dos imágenes Web GUI y Web API. O bien, se pueden poner en común en la capa inferior. Si es suficiente para el proceso con añadir los archivos las unas a las otras durante la fusión de las capas, entonces sería necesario crear varias capas para las diferentes configuraciones del servidor Apache, lo que haría perder todo el beneficio de la puesta en común.

Pero los sistemas de archivos en capas hacen que sea posible aplicar la única modificación del archivo en las imágenes de nivel superior y esto, sin eliminar el archivo de la capa Apache. El archivo simplemente se machaca con las modificaciones específicas:



Más adelante, cuando se estudie en detalle las imágenes, veremos que la gestión de las diferentes versiones, unido a esta noción de fusión, permite establecer verdaderas arborescencias de imágenes que comparten al máximo los archivos comunes, optimizando de esta manera mucho el uso de recursos.

### **e. Última capa en modo escritura**

El tercer nivel de sofisticación en la gestión de archivos, se manifiesta cuando la ejecución de los contenedores añade modificaciones al sistema de archivos. En el ejemplo utilizado anteriormente, los servidores web pueden crear archivos en el directorio `/tmp`, y el contenedor de base de datos va a escribir en `/var/lib/pgsql/data`, que es la ubicación en la que PostgreSQL sitúa por defecto sus archivos de datos.

La implementación de una capa adicional por Docker, es lo que permite gestionar estos supuestos:



Durante la ejecución de los contenedores, todas las modificaciones realizadas sobre el sistema de archivos se dirigirán a una capa aparte lo que permite, de una manera muy

sencilla, elegir si las modificaciones se deben abandonar o conservar durante el cierre del contenedor.

En el caso de los servidores web, donde los archivos temporales se pueden purgar sin problema, la superposición simplemente se abandona. Entonces calificamos el servidor con el término inglés *transient*, cuya traducción más aproximada podría ser "transitorio". Veremos a continuación cuál es el comportamiento estándar de Docker cuando un contenedor se detiene.

Por supuesto, en el caso de una base de datos, como el tercer contenedor en nuestro ejemplo, se deben conservar las modificaciones de datos realizadas. Por lo tanto, es primordial - como mínimo - transformar la capa temporal transformada en una capa de archivos definitiva. En la práctica, veremos más adelante que existen medios más adaptados para gestionar este supuesto.

## **f. Tecnologías utilizadas**

En las secciones anteriores, se ha hablado de manera genérica de sistemas de archivos por capas. La razón por la que el nombre de la tecnología subyacente no se ha utilizado, es que todavía está lejos de ser congelada.

A día de hoy, Docker utiliza UnionFS pero existen muchas alternativas como AuFS (*Another Union File System*), Btrfs (*B-tree file system*), zfs, overlay, overlay2 o Device Mapper. Algunos sistemas de archivos no se soportan, en función de la distribución utilizada. En la actualidad, se están haciendo numerosos esfuerzos para adaptar otros sistemas de archivos incrementales. Por lo tanto, no vamos a detallar este dominio cambiante. Además, el interés para el usuario final de Docker no es inmediato. Si la elección de un filesystem particular responde a una necesidad concreta o el rendimiento de la escritura de archivos es de una importancia capital para un escenario dado, visitar [la página https://docs.docker.com/engine/userguide/storagedriver/selectadriver/](https://docs.docker.com/engine/userguide/storagedriver/selectadriver/) le permitirá encontrar la configuración óptima para el sistema de archivos utilizado.




## Aspectos adicionales de Docker

La lista de los fundamentos tecnológicos sobre los que se basa Docker es, como se ha visto en las páginas anteriores, particularmente amplia. Esto no significa que Docker se contente con agrupar módulos existentes.

En particular, Docker no es un nuevo LXC: se ubica por encima de la gestión de contenedores para simplificarla y, de esta manera, realizar una verdadera virtualización de aplicaciones. Aquí se trata de una fuerte apuesta, que se traduce en la restricción de situar una única aplicación en un contenedor.

Por supuesto, esto presenta límites porque por ejemplo, las aplicaciones que necesitan tareas desencadenadas regularmente por un proceso `cron`, no se podrán desplegar en Docker sin modificaciones. Pero esta también es la razón por la que Docker ha tenido éxito allá donde LXC no tuvo el éxito masivo esperado: el enfoque "un contenedor - una aplicación" se basa en una simplificación que ha hecho que para muchos usuarios más sea abordable este enfoque de los contenedores Linux.

 En teoría, Docker no impide realmente que varios procesos funcionen al mismo tiempo en el mismo contenedor pero en la práctica, las herramientas se hacen de manera que es difícil proceder de otra manera. Hay enfoques que permiten eludir esto (en particular se detallará `supervisor`), pero contravienen el enfoque de servicios autónomos y al final, normalmente presentan problemas en materia de explotación. El enfoque por contenedores tiene valor en sí mismo, pero también muchas ventajas, debido a que obliga a separar las aplicaciones en pequeños módulos mucho más fáciles de gestionar que un gran conjunto.

Docker también gestiona una gran parte de la complejidad inducida por el enfoque por contenedores. En particular, Docker se encarga de redirigir el flujo estándar, de manera que el usuario pueda manipular el contenedor a través de un terminal interactivo, como lo haría cualquier otro proceso.

Docker también permite recuperar los logs del proceso, redirigir los puertos de red, compartir fácilmente directorios, etc. En resumen,

se comporta como una superposición que hace los contenedores Linux sencillos de utilizar y explotar.

## El ecosistema Docker

En una arquitectura virtualizada como VMware o Hyper-V, el hipervisor es central, pero tendría una utilidad finalmente reducida sin la virtualización de las redes, el administrador de máquinas virtuales, las herramientas de movimiento en caliente de máquinas, los API de operación y los clientes asociados, etc.

Sucede lo mismo con Docker, que no se usa solo. Respecto a su función central de gestión del ciclo de vida de los contenedores, es la herramienta más vista, pero no muestra todo su potencial excepto si se une a un ecosistema cada vez más nutrido.

Antes de todo, Docker es múltiple y se compone en sí mismo por un cliente, un API y de un servidor en forma de demonio. Pero la funcionalidad de almacén central realizado por Docker Hub, también es principal: un almacén público, accesible con una sencilla conexión a Internet, permite favorecer la reutilización de las imágenes ya preparadas. Esta es una de las razones principales de la velocidad de adopción excepcional de Docker.

Desde el inicio, Docker ha sido capaz de integrar tecnologías desarrolladas de manera externa inicialmente, para ofrecer funcionalidades adicionales necesarias para su integridad.

De esta manera, la herramienta Fig rápidamente se convirtió en Docker Compose y ha pasado por varias versiones desde entonces. Automatiza la implantación de varios contenedores relacionados los unos con los otros. En nuestro ejemplo de aplicación 3-tiers presentado anteriormente, un script Compose se ocuparía de arrancar los tres contenedores y conectarlos. Utilizaremos Docker Compose en los ejemplos de implantación, un poco más adelante en el libro, y se explicará la mayor parte de sus comandos principales.

Docker Machine permite provisionar máquinas de despliegue de manera unificada, sea cual sea el soporte utilizado, local o en cloud. Permite reservar rápidamente las máquinas para desarrollar contenedores en Amazon, Azure, etc.

Para terminar, Docker Swarm permite administrar un cluster de máquinas Docker, lo que hace que aparezca como una única máquina preparada para recibir múltiples contenedores.

Usamos estas tecnologías para los ejemplos más complejos de los siguientes capítulos, pero no entraremos en el detalle de su funcionamiento.

Para terminar, Docker Notary, Docker Secrets, Docker Cloud, Docker Registry y otras muchas tecnologías, añaden funcionalidades periféricas al núcleo de Docker, a saber, el cliente, el servidor y su API, permitiendo utilizar el conjunto de manera óptima para escenarios de despliegue cada vez más complejos.

## Arquitecturas de servicios

Todas las secciones anteriores de este capítulo de introducción, sirven para colocar Docker en el paisaje informático, explicar en qué se basa, su modo de funcionamiento y su ecosistema. En esta sección se plantea la pregunta del por qué. ¿Para qué sirve finalmente Docker? Porque esta capacidad de ahorrar recursos, poner en común las capas y sellar los contenedores ligeros, ¿es realmente importante en la práctica? En pocas palabras, ¿cuál es la necesidad práctica importante que ha hecho que Docker sea imprescindible?

Hay numerosas razones para la explosión del uso de Docker en estos últimos años, pero la mayor parte de estas razones se agrupan alrededor del hecho de que al final, favorecen la aparición de arquitecturas de servicios.

### 1. Histórico de las arquitecturas de servicios

#### a. Aspectos principales

La mayor complejidad cada vez de los SI, ha sacado a la luz dos principios fundamentales.

El primero es que el valor de un SI reside más en sus interacciones que en sus aplicaciones. Una aplicación de caja registradora no sirve para mucho, si no está relacionada con la contabilidad. El interés de una aplicación de gestión de pagos se ve limitado de manera importante, si no puede enviar solicitudes de transferencias bancarias. Un software de diseño asistido por ordenador no está correctamente adaptado si no se puede asociar a las líneas de producción basadas en máquinas controladas digitalmente.

El segundo principio - que no está relacionado con la informática - es que es importante dividir para ganar. A partir de una determinada complejidad, no es realista querer gestionar un sistema con una única aplicación y la especialización implica obligatoriamente una división de las responsabilidades.

El caballo de batalla de la arquitectura de los SI, ha sido empujar este segundo principio al máximo con diferentes enfoques, durante los últimos veinte o treinta años que han conocido más o

menos éxito. No mencionaremos los primeros métodos artesanales de intercambio de datos directamente a nivel de los Sistemas de Gestión de Base de Datos con, en sus últimas evoluciones, la implantación de ETL (*Extract Transform Load*). Se trataba de los ancestros de las arquitecturas basadas en servicios, en la época donde únicamente una inter-operatividad de los datos parecía un enfoque suficiente. El fuerte acoplamiento que estos métodos imponen conduce a su desaparición progresiva.

## **b. Enfoque EAI**

La EAI (*Enterprise Application Integration*) estuvo un tiempo de moda, antes de que nos diéramos cuenta de que su dependencia con un único punto central, planteaba problemas de rendimiento de carga y fiabilidad. Se habla del síndrome SPOF (*Single Point Of Failure*), para describir este peligro de la dependencia con un servicio central, que hace de intérprete entre los demás.

## **c. Enfoque SOA**

El objetivo de los enfoques SOA (*Service Oriented Architecture*) era eliminar este problema, basándose en los middlewares distribuidos y no centralizados, normalmente las ESB (*Enterprise Service Bus*) para el transporte y las normas y estándares resultado de Internet para las gramáticas, que debían ser comprendidas por todos los participantes del "bus de servicios".

Estos enfoques han fracasado por la complejidad (utilización masiva de SOAP y de las normas WS-\*, que añaden mucho vocabulario - y por tanto pesadez - a los servicios web), así como porque se basan en normas que eran puramente técnicas. Evidentemente es útil entenderse en un intercambio en formato XML, pero si este intercambio no está contractualizado por un upstream XMLSchema establecido, no se puede construir el SI. Es necesario ponerse de acuerdo en una definición conceptual de las entidades de negocio que se realizarán, de manera que se garantice que los servicios conseguirán entenderse. El formalismo técnico utilizado para la representación y el protocolo utilizado para los intercambios, finalmente no son muy importantes. Hay herramientas de mediación eficaces, basadas en el estándar EIP (*Enterprise Integration Patterns*), que se encargan de manera eficaz de esta parte del problema.

## **d. Microservicios**

Las arquitecturas SOA podrían tener en los próximos años una segunda oportunidad, debido a unión de dos evoluciones técnicas. La primera es que las normas de negocio comienzan a aparecer en la informática y los intercambios a nivel mundial afectan a los protocolos internos de las empresas. Las aseguradoras han normalizado sus intercambios con ACORD (*Association for Cooperative Operaciones Research and Development*), las agencias de viajes por OTA (*OpenTravel Alliance*), los operadores de telefonía por eTOM (*Enhanced Telecom Operaciones Map*). La segunda es que los enfoques REST (*REpresentational State Transfer*), más sencillos que SOA, permiten una mayor difusión de los servicios en forma de APIs reutilizables fácilmente por los técnicos.

Junto a una descomposición todavía más importante de las diferentes responsabilidades, esta mezcla es la que ha impulsado el movimiento de los micro-servicios. Las arquitecturas de micro-servicios consisten en utilizar esta normalización y simplificación para llevar esta descomposición todavía más lejos, estableciendo el principio de implantación uno por uno, entre las aplicaciones y los servicios.

Recientemente, las aportaciones de las webhooks como mecanismo sencillo y estándar de interoperabilidad entre diferentes servicios en línea, así como el mejor uso de los mecanismos web, han permitido establecer enfoques cualificados de Web Oriented Architectures.

## **e. Relacionado con la construcción de los SI**

Todos estos esfuerzos de arquitectura de los SI podrán parecer desmesurados, si no se enmarcan en el contexto de la construcción de los SI. El enfoque técnico de SOA / WOA (*Web Oriented Architecture*) y de los micro-servicios, se justifica por su efecto sobre la flexibilidad del SI en su conjunto y su capacidad reforzada para alinearse más rápido con los cambios en el negocio que sostiene el SI (cambio de proceso, organización, personas encargadas de las operaciones, fusión y adquisición de entidades, etc.).

La construcción del SI, es decir, la posibilidad de aprovecharse del valor de la informática de manera continua, es el gran objetivo último de todas estas etapas de estructuración, de las que Docker es una de las vías. Por otra parte, su aporte es tal que casi se podría considerar en cierto aspecto, como el aceite que hace que las ruedas (los servicios en nuestra metáfora) giren de manera más eficaz.

## 2. Arquitectura de micro-servicios

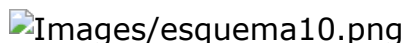
### a. Aspectos principales

Se puede retomar el ejemplo de la aplicación 3-tiers mostrada anteriormente. Se puede dividir esta aplicación en modo micro-servicios, haciendo aparecer las diferentes responsabilidades (se trata del término dedicado en la jerga de la planificación) que incorpora.



Por supuesto, solo se trata de un ejemplo. Las responsabilidades varían de un dominio de negocio a otro. Lo principal es ponerse de acuerdo en un vocabulario y los servicios comunes, que no cambian de una aplicación a otra.

La descomposición de un SI en su conjunto (es importante abstraerse del aspecto de la aplicación), en servicios autónomos y lo más débilmente acoplados posible, es un arte y justifica la presencia de expertos en planificación en los equipos de desarrollo, apoyando a los arquitectos de software. El primer paso en este camino es establecer un Plan de Ocupación de los Límites según las cuatro capas dedicadas, es decir, los procesos, las funciones, las aplicaciones y la infraestructura (hablamos del "diagrama de cuatro capas" del Sistema de Información), esquematizado anteriormente.



### b. Ventajas

Las ventajas asociadas a las arquitecturas en micro-servicios son enormes. La alineación del SI a los negocios, flexibilidad evolutiva agregada y el hecho de realmente poder sustituir una pieza del SI por otra con impactos limitados, son aportaciones que explican por sí mismas por qué, a pesar de decenas de años fracasos de los enfoques EAI y SOA, los arquitectos continúan avanzando en las arquitecturas de servicios.


Es innegable que estos factores de mejora de los SI son, una vez dominados, ventajas competitivas enormes.

La adaptación del SI al negocio consiste en reflejar en su organización informática (red, máquinas, así como software y




organigrama), el funcionamiento de una empresa o cualquier otra entidad responsable del SI. De esta manera, los cambios operativos se traducen fácilmente en modificaciones de la informática. Esta adaptación estaba particularmente de moda en los años 2000, cuando las operaciones de fusión/adquisición entre start-ups eran muy frecuentes.

Para tomar un ejemplo, imaginemos que una empresa A compra una empresa B. Si el sistema informatizado de pago de B necesita que todas las personas se introduzcan directamente en la aplicación y A no utiliza la misma, va a ser necesaria una operación manual para añadir todos los empleados de B en el sistema de pago de A. Y esta operación se deberá repetir en el resto de aplicaciones en común (por ejemplo, la gestión de tarjetas de acceso), ralentizando mucho el proceso de fusión humano de las competencias, por razones basadas en la informática.

 Images/esquema11.png

Al contrario, si los SI de A y de B estuvieran bien alineados con el funcionamiento de las empresas, sería lógico que cada una de ellas tuviera un directorio de empresa (normalmente LDAP o *Lightweight Directory Access Protocol*) y que los dos sistemas de pagos estuvieran en las ramas del directorio respectivo, utilizando el protocolo LDAP. En este segundo caso, la fusión será particularmente sencilla, porque B simplemente va a exportar sus empleados (en formato estándar LDIF, o *LDAP Data Interchange Format*, fácilmente comprensible por cualquier directorio LDAP) y A va a incorporar estas personas en su directorio, a través de una importación del archivo LDIF. Entonces, aparecerán automáticamente en el sistema de pagos, así como en el resto de servicios de A situados en las ramas del directorio de personal.

 Images/esquema12.png

En nuestro ejemplo, solo se han mostrado dos aplicaciones, pero es habitual en los SI industriales que este número se incremente hasta llegar a centenares, incluso en los casos más extremos, millares. Por lo tanto es imprescindible asignar la responsabilidad correcta, al servicio correcto: el papel del servicio del software de pagos es emitir las órdenes de pago de la nómina, no gestionar empleados de la empresa que están inscritos.

El enfoque urbanizado consiste en confiar la responsabilidad correcta al servicio apropiado y si es posible, reducir este reparto

una por una, disminuyendo así el nivel de acoplamiento de las aplicaciones las unas con las otras, facilitando al final la evolución progresiva del SI a lo largo del tiempo. En un SI correctamente urbanizado, nunca se habla de "Big Bang".

De una cierta manera, las arquitecturas de micro-servicios están alcanzando el Santo Grial de la informática, a saber, la reutilización, sucesivamente prometida desde hace decenas de años por las rutinas, librerías, componentes, componentes distribuidos, servicios web, pero nunca alcanzada, porque hasta ahora se han considerado desde un prisma únicamente técnico. La urbanización o enfoque constructivo, regula de una vez por todas el problema, aplicando el único punto de vista que prevalece a largo del tiempo, es decir, el usuario del API o del servicio. Impone un enfoque de división de los conceptos informáticos radicalmente diferente del que se ha utilizado de manera masiva hasta nuestros días, que está relacionado con criterios técnicos. La adaptación pasa por una división de las funciones informáticas controlada por el negocio, como propone Eric Evans en su enfoque Domain Driven Design, por ejemplo.

### **c. Inconvenientes**

Con tales ventajas, es legítimo preguntarse por qué no están todos los SI contruidos ya con el enfoque urbanizado, especialmente si no hay dificultades técnicas que lo impidan. La razón principal tiene que ver con el coste.

Por supuesto, hay un coste asociado a las herramientas que apoyan estas arquitecturas. Las middlewares han sido caras durante mucho tiempo. Los EAI atraen al cliente por un bajo coste inicial, pero cada conector adicional aumentaba el coste total. Los enfoques ESB necesitaban una implementación costosa y las herramientas propietarias representaban una inversión fuerte. La aparición de los ESB open source, a la vanguardia desde que la solución de la fundación Apache (ActiveMQ para el Message Oriented Middleware, Camel para el framework de mediación compatible Enterprise Integration Patterns, Karaf para la carga dinámica), cambió el juego, abriendo la puerta de los ESB a los PME.

Por supuesto, la verdadera reducción del coste de las arquitecturas de servicios ha venido del desarrollo de normas internacionales a las que se ajusta el software cada vez más. En el ejemplo anterior, es evidente que la implementación de un directorio centralizado es una inversión, justificable por un número elevado de aplicaciones que lo utilizan. Pero cuando

estos directorios LDAP existen ya en open source y sobre todo, cuando casi todos los software de gestión son capaces de colocarse en cualquier directorio, justamente gracias a la norma LDAP, el sobrecoste casi es inexistente. El extra del análisis ya se tenía en cuenta, por lo que eventualmente falta el sobrecoste de utilizar varias máquinas diferentes si queremos aislar correctamente los servicios.

Y ahí es donde recurrimos al límite principal - hasta ahora - de los enfoques de micro-servicios: para no sustituir el acoplamiento al negocio por un acoplamiento técnico, conviene disponer estos pequeños servicios en entornos sellados. Además, las arquitecturas de micro-servicios despliegan normalmente decenas, incluso centenares de servicios e incluso más entornos separados, cuando queremos gestionar de manera más fina la carga de cada uno.

Por tanto, evidentemente es posible dedicar una máquina virtual para cada uno de estos micro-servicios. Y es ahí cuando volvemos a Docker, porque nos va a permitir establecer este tipo de micro-división, por supuesto soportando al aumento de recursos necesarios.

### 3. Qué aporta Docker

La aportación de Docker a las arquitecturas de micro-servicios, es absolutamente fundamental. No es razonable pretender que estas últimas no hayan sido implementadas nunca o en cualquier caso, hayan quedado relegadas a despliegues de nicho, si las tecnologías de contenedores no hubieran sido suficientemente simplificadas. Más allá del hecho de que Docker haya normalizado el uso de los contenedores Linux (y están en el origen de su existencia en Windows), se le puede atribuir una gran responsabilidad en el hecho de que las arquitecturas de servicios estén en camino de convertirse en una realidad.

De hecho, la técnica y el método están tan relacionados que es difícil decir si las arquitecturas de micro-servicios han hecho emerger la necesidad de una tecnología como Docker, o si la puesta en disposición de Docker ha participado en la aparición de las arquitecturas de micro-servicios.

Sin Docker, las consideraciones de coste impedirían un buen número de divisiones en servicios, que tendrían sentido en términos de separación de responsabilidades. Si hiciera falta añadir una máquina virtual con todo su sistema operativo y algunas veces las licencias asociadas cada vez que se divide un

servicio en dos, las arquitecturas urbanizadas estarían totalmente limitadas, y la flexibilidad del SI estaría incompleta. Como hemos visto antes, la aportación de valor de la informática pasa por una adaptación rápida del SI con el negocio y esta rapidez, depende del nivel de separación de los servicios.

No es casualidad que empresas como Netflix, en la cima de las arquitecturas de micro-servicios, hayan conseguido un crecimiento extremadamente rápido utilizando masivamente Docker.

## 4. Hilo conductor

Después de esta larga introducción a Docker y a sus usos, en los siguientes capítulos vamos a detallar los métodos de implantación. Para esto, nos serviremos del ejemplo de arquitectura de micro-servicios como hilo conductor. Esta es la arquitectura que modelizaremos en forma de contenedores, usaremos para difundir las imágenes, integraremos en el ALM (*Application Lifecycle Management*), desplegaremos en un cluster y administraremos gracias al ecosistema Docker.

El enfoque utilizado basado en ejemplos no solo facilitará la comprensión de las funcionalidades de Docker, mostrando las razones por las que existe, sino además permitirá situar Docker en el movimiento de fondo de los sistemas informáticos que existen en este momento, de los que él es uno de los actores principales.

## 2. PRIMEROS PASOS

### Instalación de Docker


Siempre que utilice una versión lo más reciente posible de Linux, la instalación de Docker es un ejemplo de simplicidad. El objetivo de la presente sección no es detallar el proceso para todas las distribuciones de Linux, ni entrar en las opciones complejas de cuota de memoria o configuración de seguridad, sino simplemente dar las instrucciones básicas para que el lector pueda realizar los ejemplos que siguen a continuación.

La página dedicada a la instalación de Docker en el sitio web de referencia <https://docs.docker.com/engine/installation/> es exhaustiva sobre este tema, en particular con las opciones sobre las diferentes distribuciones Linux, los modos de implementación en las plataformas cloud, etc. Nuestro objetivo en esta sección, únicamente es hacer que el lector que quiera utilizar una máquina remota o bien instalar Docker en su sistema operativo Linux o Windows, encuentre rápidamente el método más estándar para hacerlo.

A partir de ahora, el producto está disponible en varias ediciones. Se dedicará una explicación para resumir las principales diferencias. El modo de actualización de Docker es una opción a determinar durante la instalación. Los diferentes canales propuestos por Docker para esta operación, también se explicarán.

### 1. Utilizar máquinas pre-configuradas

Si el lector dispone de una cuenta en el cloud, la manera más sencilla es reservar una máquina que soporte Docker en este tipo de plataformas. De esta manera, la instalación ya está realizada, actualizada y en una configuración que podemos esperar que sea óptima.

 Casi todas las plataformas ofrecen suscripciones gratuitas para un periodo de prueba, incluso ofertas sin gastos en las que solo está limitado el consumo de recursos. Para formarse en una tecnología, estas soluciones son ideales, porque permiten no tener que modificar nada en su ordenador personal, ni perder tiempo en los aspectos relativos a la instalación o los requisitos

previos (que no tienen ningún valor de aprendizaje, porque son completamente diferentes en otro contexto).

Una vez más, el objetivo del presente capítulo no es describir de manera exhaustiva los métodos para lanzar Docker en los clouds existentes. Se dará un ejemplo sobre Microsoft Azure.

Conéctese al portal de Azure (<http://portal.azure.com>).

Pulse en **Crear un recurso** en la parte superior izquierda.

Acceda a Marketplace.

Inicie una búsqueda con "docker".



La lista con los resultados da una idea del número de posibilidades existentes, para disponer de un Docker en Azure. Algunas máquinas están dedicadas a Docker, otras están editadas por Docker para soportar servidores de aplicaciones (por ejemplo PostgreSQL), algunas ofrecen un sistema operativo Linux con Docker preinstalado (como Docker en Ubuntu Server, ahora conjuntamente por Canonical y Microsoft).


Esta primera lista es la oportunidad de explicar las diferentes ediciones posibles para Docker. Desde 2017, el editor Docker Inc. ha elegido ofrecer dos ediciones de Docker.

La primera, llamada Community Edition, que reúne las funcionalidades tradicionales de Docker, tal y como se distribuía previamente: se trata de una versión gratuita que no incluye soporte, pero que permite lanzar contenedores, gestionarlos y vincularlos por la red. A groso modo, se trata de las tecnologías Docker Engine, Docker Compose y Docker Swarm y es la edición que conforma el tema de estudio de este libro.

La segunda, llamada Enterprise Edition, contiene todas las funcionalidades de la edición Community, pero añade funcionalidades de más alto nivel, así como el soporte. Se dirige claramente a las empresas que desean obtener una garantía (incluso pagando) de funcionamiento continuo y servicios avanzados, como la certificación de la infraestructura, un administrador privado para las imágenes Docker, etc. No se van a cubrir estas funcionalidades avanzadas en este libro, cuyo objetivo es alcanzar un conocimiento profundo de los conceptos básicos de Docker y no tener una idea de todo el ecosistema. La edición Enterprise se ofrece en forma de tres sub-ediciones: Basic,

Standard y Advanced, en orden creciente de riqueza funcional (y por lo tanto, de tarifa).

Para continuar centrados en un enfoque ligero, modifique la búsqueda en el portal Azure para sustituir la palabra clave "docker" por "coreos".

 CoreOS es un sistema operativo extremadamente compacto y hecho a medida, alrededor de Docker. Permite disponer de una base muy ligera para ejecutar Docker y puede servir como imagen básica ligera para las imágenes Docker. Estos conceptos de imagen y de "peso" de estas últimas, se explicarán en detalle en los siguientes capítulos.

En la lista que aparece, seleccione una de las opciones posibles (la release estable es la elección más razonable), para la máquina CoreOS a crear.

 images/2.png

Pulse en **Crear**.

Asigne un nombre a la máquina.

Seleccione un nombre de usuario (utilice "core").

Para una sencilla máquina de pruebas, es inútil establecer una autenticación por clave SSH (*Secure Shell*). Una contraseña es suficiente.

 images/3.png

Pulse en **OK**.

Para la segunda etapa del asistente, seleccione una capacidad de máquina virtual (para nuestra prueba, una máquina muy pequeña será suficiente, por ejemplo una DS1 Standard).

 images/4.png

Pulse en **Seleccionar**.

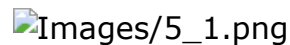
La siguiente etapa del asistente permite elegir valores de opciones avanzadas. En caso de una utilización únicamente para probar los ejemplos del libro, se pueden desactivar las opciones de "managed disks" y diagnóstico de boot. Al contrario, la opción de apagado automático diario de la máquina pueda ser interesante: ya que no habrá ninguna funcionalidad de producción en esta máquina, la seguridad respecto a la facturación no se compensa por el riesgo de un defecto de servicio. Esta opción es muy importante en este tipo

de máquinas, para no encontrarse con una facturación de varios centenares de euros al cabo de varios meses porque se ha olvidado de la máquina, lo que tiene más opciones de suceder con una máquina que solo se utiliza para pruebas.



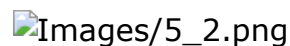
Pulse en **OK**.

Valide el resumen, acepte las condiciones de utilización y pulse en **Adquirir** para lanzar la creación de la máquina virtual.

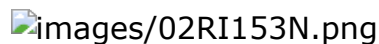


Al cabo aproximadamente de un minuto, la máquina está preparada y la interfaz del portal Azure muestra una notificación, así como las propiedades del grupo de recursos creado al mismo tiempo que la máquina virtual.

Pulse en **Connect** para obtener la información de conexión.



Lance PuTTY, que vamos a utilizar para conectarnos a la máquina.



Después de haber introducido la dirección IP (y eventualmente guardado la sesión con un nombre concreto, como en la captura anterior), pulse en **Open**.

PuTTY avisa de que la clave todavía no se conoce ni se ha aceptado. Por tanto requiere una validación por parte del usuario para continuar la conexión:



Pulse en **Sí** para dar esta aprobación.

Para conectarse, use el usuario y la contraseña elegidos durante la creación de la máquina virtual en el portal Azure:

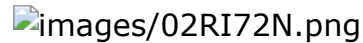


Ahora ya están disponibles las funcionalidades de Docker.


*Visualizar la información de la versión de Docker instalada*



`docker version`



El hecho de que el comando `docker` responda, muestra que la herramienta se ha instalado correctamente por defecto en la distribución CoreOS, de la que hablaremos un poco más adelante.

 En un entorno profesional, se recomienda utilizar la autenticación por certificado. En el caso de nuestro ejemplo sobre Azure, aconsejamos <http://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-use-ssh-key/>, que explica de manera muy detallada la manera correcta de proceder.

Antes de continuar con el uso de la herramienta, vamos a mostrar las alternativas que nosotros mismos instalaremos.

## 2. Instalación de Docker en Linux

Como se ha explicado al inicio del capítulo, existen numerosas maneras de instalar Docker en Linux y las opciones no faltan. La premisa de partida de este libro es no detallar estas operaciones, que recaen en más de una competencia del administrador de sistemas. El libro se centra en ofrecer un conocimiento profundo de los principios de funcionamiento de Docker.

Por lo tanto, nos limitaremos a mostrar las dos maneras más estándares de hacerlo, para que el lector pueda establecer rápidamente un Docker en su máquina Linux y continuar la lectura del libro con lo necesario para realizar los ejercicios.

Ubuntu era una distribución muy utilizada. La versión más reciente en el momento de escribir este libro, es la que se utilizará para mostrar los dos modos de instalación (configurando los almacenes y por script). Se utilizará la edición Community. Ninguno de los ejercicios que siguen a continuación, tendrán nada que ver con las funcionalidades de la edición Enterprise.

### a. Requisitos previos de sistema

En las versiones recientes, la mayor parte de los requisitos previos de software forman parte de la distribución. De hecho, los

dos puntos principales que hay que tener en cuenta son un nodo reciente (3.10) mínimo y una instalación en 64 bits.


## **b. Instalación por administrador de paquetes**

El método más sencillo consiste en utilizar el administrador de paquetes APT para instalar Docker. A partir de ahora, este último se incorpora en la distribución.

Docker (o bien su mecanismo de instalación), requiere que haya algunos paquetes en el sistema. El siguiente comando permite garantizar su instalación:

### *Instalación de los requisitos previos de software con APT*

```
sudo apt-get update  
sudo apt-get install apt-transport-https ca-certificates curl  
software-properties-common
```

 Si alguno de estos requisitos previos de software ya está presente en su sistema, puede ajustar el comando para no instalarlos.

En un sistema actualizado puede que reciba los siguientes mensajes, que informan al usuario de que no es necesario modificar nada:

- apt-transport-https is already the newest version (1.4).
- ca-certificates is already the newest version (20161130).
- software-properties-common is already the newest version (0.96.24.13).
- curl is already the newest version (7.52.1-4ubuntu1.1).

La siguiente etapa consiste en añadir la clave que permite autorizar al mecanismo de gestión de paquetes para que acceda al almacén Docker. Los paquetes no están en los almacenes estándares.

### *Añadir la clave para el almacén Docker*

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |  
sudo apt-key add -
```


La seguridad es un asunto que concierne a todos. Es importante verificar la huella de esta clave. En caso de que la descarga

hubiera comprometida por un tercero, su resolución de dominio podría haber sido pirateada, etc.


### Visualización de la huella de la clave de almacén

```
sudo apt-key fingerprint
```

La visualización como resultado de este comando se debería parecer al siguiente y, en particular, la huella debería ser 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88 para la entrada relativa a Docker CE:

images/02RI73N.png

Si este valor es diferente, la instalación tiene muchas opciones de haber sido comprometida; en ese caso, la clave se debe eliminar inmediatamente para no recuperar por error los paquetes comprometidos del almacén probablemente pirateado, y debemos tomar las medidas necesarias para eliminar estos riesgos, tanto para usted como para su entorno profesional o personal.

 En informática, sobre todo en la actualidad, una cierta dosis de paranoia es más una cualidad que un defecto. No debe fiarse de este libro para ofrecerle la huella correcta, vaya a comprobarla usted mismo en el sitio web de Docker (el autor no ofrece la URL, porque solo introduciéndola usted mismo se asegura de no ser redirigido a un sitio web malicioso que confirmaría una clave falsa, con toda la apariencia del sitio web Docker real).

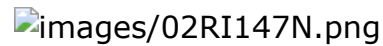
Si la clave registrada era la correcta, el siguiente comando consiste en añadir el almacén propiamente dicho a la lista de almacenes reconocidos por el sistema. El comando a utilizar en un sistema de tipo x86 o AMD en 64 bits (en pocas palabras, la gran mayoría de las máquinas para los potenciales usuarios de Docker), es el siguiente.

### Añadir el almacén Docker

```
sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

La utilización de la palabra clave "stable" corresponde al modo de actualización deseado. Docker propone dos modos, a saber, estable y edge. En el primero, las actualizaciones están a disposición de manera trimestral y el contenido tiene garantía de

estabilidad. En el segundo, las actualizaciones son mensuales, por lo que el usuario se beneficia más rápido de las nuevas funcionalidades. O según palabras de Docker:



Atención, no confunda este segundo modo con el modo experimental de funcionamiento de Docker. El modo experimental se puede activar cambiando una opción de inicio del demonio Docker, haciendo de esta manera que todas las funcionalidades nuevas queden activadas, incluso durante el desarrollo, lo que puede afectar al correcto funcionamiento de Docker. Por lo tanto, este modo se debe reservar en general a los beta-testers y desarrolladores. El modo de difusión edge no va tan lejos: ofrece alguna de las funcionalidades más recientes, que se podrán beneficiar de varios meses en producción, pero son funcionalidades probadas y validadas antes de su puesta en disposición por el canal de actualización oficial, aunque no es el que ofrece la mayor garantía de estabilidad.

Si se presenta la duda de qué canal elegir, lo más sencillo es utilizar el canal estable.

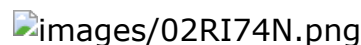
Toda esta preparación permite introducir comandos que van a instalar Docker Engine de manera efectiva.


### Instalación del paquete Docker CE

```
sudo apt-get update  
sudo apt-get install docker-ce
```

Puede que sea necesario confirmar la descarga, introduciendo la letra "y" seguida por la tecla [Intro].

Al cabo de algunos segundos, en función de la velocidad de conexión a Internet, se instalará Docker, lo que se puede probar escribiendo el comando `docker version`. El resultado debería ser parecido al siguiente:



 Atención, existe otro paquete llamado simplemente "docker", y que se corresponde con una herramienta homónima pero diferente a la de nuestro estudio. De la misma manera, "docker.io" ya no es el gestor de paquetes utilizado por Docker. Por lo tanto es posible instalarlo, pero las versiones no necesariamente serán consecutivas. Atención, en numerosas

distribuciones Linux, durante la entrada del comando docker en un sistema que no contiene el programa, un mensaje le puede invitar a instalar este paquete docker.io, cuando ya no es el que se debe utilizar.

### **c. Instalación por script**

En lugar de pasar por varios comandos como se ha mostrado anteriormente, es posible instalar Docker a partir de un script que se ocupa de todo, incluida la detección de la distribución utilizada. Este script se obtiene desde <https://get.docker.com> para la versión estable (o <https://test.docker.com> para el caso inverso). Puede descargarlo desde un navegador y guardarlo en un archivo o bien realizar esta operación a través de una línea de comandos.

#### *Descarga del script de instalación de Docker*

```
wget -O dockerinstall.sh https://get.docker.com
```

Echar un vistazo al archivo es muy instructivo para aprender las operaciones de instalación realizadas, y constituye también un buen hábito que se debería adquirir cuando se ejecuta un archivo descargado desde el exterior, por sencillas razones de seguridad. La siguiente operación consiste en ejecutar el archivo desde el shell de su sistema operativo.

#### *Ejecución del script de instalación de Docker*

```
sh ./dockerinstall.sh
```

La instalación en sí misma lleva algunos minutos. Como otras ocasiones, el comando `docker version` permite verificar si se ha desarrollado correctamente.

Uno de los primeros comandos que hay que implementar para no tener que usar el prefijo `sudo` en los comandos Docker, es el siguiente. Va a añadir su usuario al grupo docker, dándole de esta manera los permisos necesarios.

#### *Añadir el usuario al grupo docker*

```
sudo usermod -aG docker [identificador de la cuenta utilizada]
```

Si es necesario, puede ser útil volver a arrancar el servicio docker para que el comando se tenga en cuenta.

### *Volver a arrancar Docker*

```
sudo systemctl restart docker
```

## 3. Instalación de Docker en Windows

### **a. Una paradoja resuelta**

Docker es un producto pensado alrededor de tecnologías del nodo Linux, a saber, los controles de grupos y los espacios de nombres, cuya implementación es muy dependiente de esto. Las tecnologías citadas no existen en el sistema operativo Windows. Instalar Docker en Windows ha consistido durante mucho tiempo, simplemente en hacer funcionar una máquina virtual Linux en Windows y llevar a esta los contenedores Linux. Esto se hacía con la herramienta Boot2Docker, empaquetada bajo el nombre de Docker Toolbox, en Windows y Mac OS X.

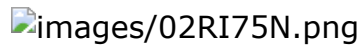
Por supuesto, este enfoque iba completamente en contra de los objetivos de ligereza de Docker y no se podía utilizar en producción. Pero en las máquinas de desarrolladores, permitía establecer rápidamente dependencias de servicios o incluso montar un contenedor para probar un código realizado en una máquina Windows pero cuyo objetivo era un despliegue en Linux (lo que sigue siendo habitual).

La normalización del API Docker y la composición de las imágenes por Open Container Initiative, ha abierto la puerta a una implementación Windows nativa y que sería totalmente compatible con la referencia en Linux. Sin embargo, seguía siendo el camino más duro de recorrer, es decir, encontrar cómo implementar las funcionalidades de tan bajo nivel en Windows, limitando los impactos.

Los equipos de desarrollo de sistemas de Microsoft en algún momento han pensado en utilizar la tecnología Drawbridge de aislamiento de procesos de Azure, pero finalmente se ha abandonado a favor de tecnologías más antiguas pero mejor adaptadas a la necesidad, a saber, Windows Job Object (que permite gestionar grupos de recursos) y Server Silos (tecnología de aislamiento de sistemas de archivos, registros, etc.).

Un artículo de Taylor Brown, aparecido en el número de MSDN Magazine de abril del 2017, explica en detalle cómo se ha realizado la implementación, pero quizás sea más importante saber que la implementación Windows de Docker se ha realizado

por Microsoft, en estrecha colaboración con el equipo Docker. Por ejemplo, se pueden ver numerosas secciones de código fuente en el almacén Docker en GitHub para realizar el soporte de Windows, como por ejemplo, el commit seleccionado en la siguiente captura de pantalla:



De hecho, una particularidad del sistema operativo de Microsoft es que hay dos Windows. Lo que el gran público llama de manera tradicional Windows es decir, la versión para el puesto cliente. Esta es muy diferente de su hermano mayor, Windows Server, que se utiliza en los servidores. Por supuesto, la interfaz gráfica (cuando se activa del lado servidor), es la misma y comparten numerosas funcionalidades, pero el núcleo del sistema es diferente en muchos aspectos.

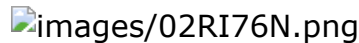
De esta manera, la implementación de Docker en Windows es muy diferente entre las dos líneas de Windows. En Windows 10, se trata de un programa interactivo que permite al usuario lanzar los contenedores Windows o Linux. En este segundo caso un soporte que vuelve al antiguo método de virtualización descrito anteriormente en este capítulo, esta vez desplegando en Hyper-V no ya una imagen boot2docker, sino una versión más reciente basada en un Moby Linux (distribución gestionada por Docker). En Windows Server 2016, el foco se pone en una implementación "in house" de la gestión de contenedores, que sigue siendo compatible con los API (gracias a la normalización), pero añade sus propias especializaciones como por ejemplo, un nivel de aislamiento adicional de los recursos por medio de una utilización reforzada del hipervisor "in house" Hyper-V.

De manera muy lógica, se ha decidido que la versión para Windows 10 incorpore una Community Edition mientras que la versión para Windows Server 2016, sería una Enterprise Edition. La diferencia de destino entre los dos sistemas operativos (desarrollador por un lado, operador de producción por otro), se corresponde perfectamente con la separación de los servicios de las dos ediciones, la primera gratuita para el desarrollo y la segunda con un soporte industrial para su puesta en producción.

## **b. Instalación en Windows 10**

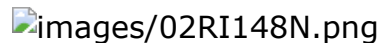
Consecuentemente a lo que se espera de una instalación en Windows (el sistema operativo cliente), la de Docker es extremadamente sencilla. En la página dedicada al sitio web de Docker, que se puede ver a continuación, un clic en el botón

correspondiente en el canal deseado inicia la descarga de un ejecutable de instalación.



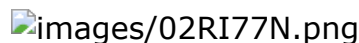
El programa Docker for Windows Installer.exe es un asistente sin características particulares y no nos detendremos aquí en explicar las etapas, porque esto no añade nada al lector. Sin embargo, hay algunos puntos que es interesante conocer.

El primero es que Docker for Windows (o Hyper-V, que es una de sus dependencias y que se activará si es necesario durante la instalación), estará potencialmente en competencia con otras aplicaciones para la utilización de la tecnología de virtualización VT-X. De manera habitual, los usuarios de VirtualBox de Oracle verán que aparece un mensaje como el siguiente, que le avisa de que su herramienta anterior de virtualización dejará de funcionar si acepta que continúe la instalación:



Por supuesto, sigue siendo posible volver a VT-X a VirtualBox, desactivando Hyper-V, pero esto se hace pagando el precio de la no disponibilidad de Docker durante el siguiente reinicio. En la actualidad, no se ha encontrado ninguna solución de compatibilidad, excepto establecer un multiboot y volver a lanzar su máquina cada vez que es necesario, lo que no es práctico. Si la portabilidad de las máquinas virtuales VirtualBox a Hyper-V es sencilla (por ejemplo, utilizando las definiciones Vagrant), podría ser una oportunidad para dar el paso.

Otra particularidad de Docker en Windows (puesto cliente), ya se ha mencionado anteriormente en este capítulo: esta implementación permite la ejecución de contenedores Windows, así como de contenedores Linux (pero no simultáneamente). De esta manera, en el menú que aparece cuando pulsa en el icono en forma de ballena correspondiente a la aplicación Docker, verá un menú que permite cambiar de un modo al otro:



Desde sus inicios, Docker se ha pensado como un conjunto cliente + servidor. El cliente Docker reúne las órdenes (normalmente la línea de comandos) y las envía al servidor, por medio de una API estándar. El servidor, por su lado, se encarga de la gestión de los contenedores y no interactúa directamente

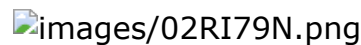


con el usuario. Este modo de funcionamiento y el hecho de que el API Docker pasa por el protocolo HTTP, permite que sea sencillo el control de un demonio Docker en una máquina, desde un cliente a otra máquina. Esta distinción del cliente y del servidor es visible en los resultados del comando `docker version` y va a permitir describir de manera más técnica la diferencia entre los contenedores Windows en Windows y los contenedores Linux en Windows.

En el segundo, es decir, cuando Docker en Windows funcione en modo "Linux containers", la salida del comando `docker version` es la siguiente:

images/7.png

La arquitectura del cliente es windows/amd64, pero la del servidor es linux/amd64. Si el usuario utiliza el menú "Switch to Windows containers", el resultado será el siguiente:

images/02RI79N.png

El cliente siempre es de tipo windows/amd64, pero esta vez el servidor también lo es. En resumen, en Docker en Windows 10, el cliente siempre es una aplicación Windows, que pueda controlar o bien una implementación nativa Windows del servidor Docker (normalmente la documentación utiliza el término inglés de Docker Engine) o bien una emulación por máquina virtual cacheada de un servidor Docker en Linux.

### **c. Instalación en Windows Server 2016**

Como en cualquier sistema operativo dedicado a la producción que se precie, la instalación de la gestión de contenedores Docker en Windows Server se realiza por la línea de comandos. Los comandos a utilizar en un shell PowerShell son los siguientes:

```
Install-Module -Name DockerMsftProvider
Install-Package -Name Docker -ProviderName DockerMsftProvider
Restart-Computer -force
```

En Windows, este no es el protocolo HTTP utilizado por defecto, sino las *named pipes*. Si es necesario acceder con un cliente cualquiera pasando por el tradicional puerto TCP 2375, entonces se deben utilizar los comandos adicionales para abrir este modo de funcionamiento:

```
Stop-Service Docker
dockerd --unregister-service
dockerd -H npipe:// -H 0.0.0.0:2375 --register-service
Start-Service Docker
```

Y por supuesto, también será necesario abrir el puerto 2375 en el cortafuegos local (así como en todo equipamiento de red adicional, por ejemplo en el Network Security Group de Azure si utiliza una máquina en el cloud de Microsoft). En Windows Server 2016 que utiliza PowerShell, el comando es el siguiente (el valor del atributo `name` se corresponde con el título de la regla del cortafuegos y por lo tanto, se puede modificar por el usuario):

```
netsh advfirewall firewall add rule name=docker dir=in protocol=tcp
localport=2375 action=allow
```

Una vez que se han lanzado estos comandos, la utilización de la línea de comandos es exactamente la misma que en Windows 10 y como en Linux, porque el API se estandariza y Microsoft garantiza que respeta este estándar en su implementación de los contenedores. Las imágenes a utilizar serán las imágenes Windows, pero esto está fuera del ámbito de este libro.

Por lo tanto, la instalación en Windows Server no es más complicada que en Windows 10. Por supuesto, también es posible (y más sencilla para las pruebas) recurrir a máquinas ya preparadas. Para esto, los botones Deploy to Azure disponibles en la documentación Docker, son particularmente prácticos. En Azure justamente, algunas imágenes Windows Server 2016 utilizan en su título el sufijo “- with containers”, lo que significa que la gestión de contenedores se activará tan pronto como termine la instanciación.

Para terminar, una última observación: el usuario se podría preguntar por qué la implementación de Docker para Windows Server no tiene el mismo comando “Switch to Linux containers” que la versión para Windows 10. Esto es porque el soporte de los contenedores Linux en Windows recurre a una máquina virtual cacheada, lo que no constituye una solución con suficiente rendimiento y probada para producción. Si se debe realizar una mezcla en producción entre los contenedores Windows y Linux, entonces se puede realizar un cluster de máquinas Docker híbrido, pero esto excede el ámbito de este libro.

## **d. Máquina virtual**

Incluso si no hay nada de particular que decir sobre esta solución, es posible simplemente instalar Docker en una

máquina virtual Linux, que se aloje en una máquina física en Windows.

Técnicamente, el resultado no es muy diferente del antiguo enfoque Boot2Docker/Docker Toolbox, pero la ventaja de este enfoque es además un mayor dominio de los diferentes argumentos en juego. La operación se parece un poco más a un entorno real de producción, sobre todo si este está virtualizado y tenemos la precaución de conectarse a la máquina virtual por SSH desde el host, en lugar de utilizar una sesión interactiva.

No vamos a detallar este modo de instalación, que consiste exactamente en una instalación en Linux, una vez que se ha creado una máquina virtual en el hipervisor elegido.


# Hello World, Docker

## 1. Puesta en marcha de un contenedor sencillo

Una vez que se ha tratado el tema de la instalación de Docker, es momento de comenzar a explorar la herramienta en sí misma. Como cualquier tecnología informática que se precie, Docker dispone de un ejemplo "Hello World". En nuestro caso, un contenedor que no hace nada salvo mostrar un mensaje de bienvenida. El interés de este tipo de enfoque es que permite validar que la instalación del producto se hace de manera correcta, verificar que el conjunto de la cadena lógica funciona y servir de ayuda al usuario principiante.

### *Poner en marcha un contenedor Hello World*

```
docker run hello-world
```

 A partir de ahora y en lo que sigue en este libro, consideramos que se trata la gestión de los permisos. De esta manera, si es necesario utilizar en los comandos la palabra clave `sudo` en Linux como prefijo, ya no lo mostraremos más en los ejemplos. Se puede recibir un mensaje con el texto " `permission denied` " o una expresión parecida, lo que indica que se debe añadir la palabra clave si es necesario o tratar el problema de seguridad. Con el mismo objetivo de simplificación, los ejemplos se muestran en Linux.

Excepto algún problema de instalación de Docker o de acceso a Internet, la visualización debería ser la siguiente:

```
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. (Assuming it was not already locally available.)
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
```

```
For more examples and ideas, visit:
http://docs.docker.com/userguide/
```

## 2. ¿Qué ha sucedido?

Aunque la ejecución del comando anterior sea muy rápida, se han producido un determinado número de operaciones para llegar a este resultado.

### a. Recuperación de la imagen

En primer lugar, Docker necesita leer la imagen llamada `hello-world`, que se le ha pasado como argumento del comando `run`.

Como se ha tratado rápidamente en la introducción, Docker dispone de un almacén de imágenes a las que se puede acceder por Internet. Este almacén, accesible a través de la URL <https://registry.hub.docker.com> y también conocida con el nombre de registro, agrupa todas las imágenes "oficiales" ofrecidas por Docker. Veremos un poco más adelante que es posible almacenar sus propias imágenes si desea compartirlas o crear almacenes privados.

Observe que la imagen ya no está presente en la máquina host, Docker la ha descargado. Es posible recuperar la imagen en primer lugar.

#### Descargar una imagen sin ejecutarla

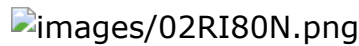
```
docker pull hello-world
```

En el caso de la imagen `hello-world`, que pesa menos de 1 KB, la diferencia de tiempo es despreciable, pero para imágenes más voluminosas puede ser particularmente útil separar la descarga de la ejecución.

Docker gestiona una caché de imágenes en la máquina y, durante la siguiente puesta en marcha de una instancia del contenedor en la misma imagen, esta no conducirá necesariamente a su recarga salvo, por supuesto, si se ha modificado entre medias.

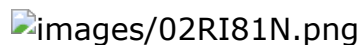
### b. Anatomía de la imagen obtenida

Una búsqueda en el registro Docker permite localizar la página correspondiente a la imagen en línea, a saber: <https://registry.hub.docker.com/u/library/hello-world/>



La página hace referencia a diferentes características de la imagen y, en particular a un archivo particularmente importante que se llama `Dockerfile`. Como Docker multiplataforma con su soporte por Microsoft en Windows desde hace poco tiempo, en el caso de Windows las dos plataformas aparecen con una característica particular que es la imagen básica. Lo que se utiliza aquí es un "nanoserver", que es más ligero que el tradicional "windowsserverbase", completo pero muy pesado. Por lo tanto, hubiera sido posible ver una tercera plataforma para la etiqueta "latest" de nuestra imagen de prueba. Pero volvamos al archivo `Dockerfile` en sí mismo. Este archivo se corresponde con la definición textual del contenido de la imagen.

Siguiendo el enlace del mismo nombre (como se ha explicado anteriormente, utilizaremos las versiones Linux para los ejemplos), se accede a la versión actual de este archivo, que se almacena en el almacén de código fuente GitHub:



El siguiente capítulo explicará en detalle la gramática de un `Dockerfile`, pero es muy sencilla. A continuación se muestra rápidamente la explicación del funcionamiento de las tres líneas:

- La primera define sobre qué imagen se basa la presente imagen. En nuestro caso, la palabra clave `scratch` se corresponde con una imagen vacía. El proceso que se ha iniciado en el contenedor funcionará directamente en el sistema operativo de la máquina host, por supuesto conservando todas las propiedades de estanqueidad aportadas por Docker.
- La segunda se asegura de que cuando se lea el archivo `Dockerfile` para generar una imagen, el contenido del archivo `hello` se añadirá a la raíz del sistema de archivos de la imagen.
- Para terminar, la tercera establece que la puesta en marcha de la imagen provocará la ejecución del comando `/hello`, que ejecutará el contenido del archivo copiado anteriormente.

Veremos más adelante cómo transformar este archivo `Dockerfile` en una imagen utilizable (gracias al comando `build`), pero para el caso presente de nuestro ejemplo de sencilla ejecución de un contenedor a partir de una imagen preexistente, Docker ya ha realizado anteriormente esta operación de compilación y el registro expone una imagen ya compilada.

Retrocediendo un nivel en la arborescencia del proyecto, encontramos el famoso archivo `hello`:

 `images/02RI82N.png`

Como información, se trata de un archivo binario que se genera gracias al archivo `Makefile`, que se puede encontrar en la raíz de la arborescencia del proyecto `hello-world` en GitHub. Este archivo se encarga de producir un archivo ejecutable a partir del código en lenguaje C almacenado en el archivo `hello.c`. Para los más curiosos, este código fuente solo contiene las llamadas de sistema de manera que se genera una imagen lo más reducida posible, siendo compatible con cualquier sistema Linux de 64 bits.

A continuación, el inicio de este archivo:

 `images/02RI83N.png`

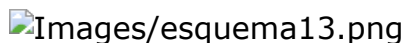
### **c. Ejecución del proceso**

Ahora que hemos visto cómo se ha proporcionado la imagen y después recuperado en la caché local, ¿qué pasa con el resto del comando `run`?

En primer lugar, Docker va a crear un contenedor, es decir, un espacio estanco en el sistema de la máquina host, con sus propios espacios de nomenclatura para la gestión de los procesos, las entradas/salidas, la red, etc.

La instanciación continúa con el montaje de un sistema de archivos en capas (como se ha explicado en la introducción), de manera que el contenedor final dispone de los archivos contenidos en la imagen, pero añadiendo también una última capa que es la única en modo escritura. Si se añaden modificaciones al sistema de archivos por medio del proceso alojado en el contenedor, esta es la capa que se modificará.

En nuestro ejemplo `hello-world`, no se realiza ninguna escritura, pero la capa existe de cualquier manera:



De hecho, la capa `scratch` no existe, porque se trata de un identificador particular que necesita que la imagen no se base en ninguna otra imagen existente.

A continuación, Docker procede a la creación de una interfaz de red que permitirá la comunicación entre el contenedor y la máquina host, ubicada en el contenedor en esta configuración de red, así como en el sistema de archivos correcto usando el comando `chroot` y puede pasar a la última etapa, es decir, a la ejecución del proceso propiamente dicha.

El proceso iniciado es el se había previsto en la imagen por el comando `CMD` del archivo `Dockerfile`, en nuestro caso `/hello`. El contenedor va a ejecutar este comando, que desencadenará el archivo ejecutable encargado de mostrar el texto de bienvenida contenido en el archivo `hello.c`. La redirección de las entradas y salidas de consola permitirá a la máquina host manipular más fácilmente el proceso aislado.

#### **d. Parada del contenedor**

La imagen `hello-world` contiene un proceso previsto para realizar una tarea (en este caso mostrar un mensaje de bienvenida) y después detenerse. En teoría, podríamos seguir el proceso con el siguiente comando.

##### Listar los contenedores en curso

```
docker ps
```

En la práctica, el contenedor `hello-world` no se había lanzado en segundo plano. El comando `run` solo devolverá el control cuando el proceso termine, por lo que es imposible lanzar el comando mientras el contenedor se ejecuta. Por otro lado, incluso si el contenedor se hubiera lanzado en este modo, la ejecución del proceso es tan rápida que devolvería el control antes de que pudiéramos escribir el comando.

Afortunadamente, es posible encontrar la traza de los contenedores ejecutados incluso después de que se detengan.



### Listar todos los contenedores

```
docker ps -a
```

El comando anterior genera la siguiente visualización:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS		PORTS	NAMES
ff37c5f56dc1	hello-world:latest	"/hello"	10 seconds ago
Exited (0) 9 seconds ago			agitated_franklin

La paginación no permite verlo fácilmente, pero se trata de dos líneas, de las que la primera ofrece los títulos correspondientes a los valores de la segunda. Volvamos a la diferente información:

- **CONTAINER ID (ff37c5f56dc1):** se trata del identificador único del contenedor. Docker crea automáticamente un nuevo identificador durante cada ejecución de un contenedor, incluso si instancia la misma imagen. Este identificador se puede proporcionar, incluido en su forma reducida (los cuatro o cinco primeros caracteres en general son suficiente), cuando queremos pausar o detener un contenedor.
- **IMAGE (hello-world:latest):** se trata del nombre de la imagen utilizada para instanciar el contenedor. El sufijo `:latest` indica que se trata de la última versión disponible. Es posible especificar una versión particular de una imagen. En nuestro ejemplo, como este no era el caso, se utiliza por defecto la más reciente.
- **COMMAND ("/hello"):** esta información retoma el contenido de la línea CMD del archivo Dockerfile, cuyo comando lanza el proceso en el contenedor.
- **CREATED (10 seconds ago):** da la fecha de inicialización del contenedor.
- **STATUS (Exited(0) 9 seconds ago):** el estado del contenedor contiene varias informaciones. En nuestro caso, como se ha dicho anteriormente, el contenedor se detiene y el proceso interno sale con el código 0, que por convención se corresponde con un desarrollo sin errores. La parada del proceso ha tenido lugar en el momento expuesto.
- **PORTS:** los puertos de red expuestos por el contenedor (ninguno en este ejemplo tan sencillo), solo muestran el texto. Volveremos sobre los casos más complejos de servidor que expone las funcionalidades en los puertos, mostrando cómo Docker los gestiona.

- NAMES (agitated\_franklin): es posible asignar un nombre a un contenedor durante su ejecución, para poder manipularlos más fácilmente que por su identificador. En ausencia de nombre concreto, Docker crea un nombre a partir de una combinación de nombres propios y adjetivos, lo que ya es un progreso respecto a ff37c5f56dc1.

Otro signo de la ejecución del contenedor, es que la imagen hello-world se ha descargado y se encuentra en la lista de imágenes presentes en la máquina host.

### Listar las imágenes

```
docker images
```

El resultado será algo parecido a:

REPOSITORY	TAG	IMAGE ID
CREATED	VIRTUAL SIZE	
hello-world	latest	05a3bd381fc2
10 days ago	1.84kB	

Incluso aquí, por el momento se muestra una línea de título y una única línea de datos, que se leen como sigue:

- REPOSITORY (hello-world): el nombre de la imagen en el almacén.
- TAG (latest): su versión, puede ser un número o una denominación textual. Veremos un poco más adelante que estos tags se pueden multiplicar respecto a una misma imagen.
- IMAGE ID (05a3bd381fc2): un identificador único para la imagen.
- CREATED (10 days ago): la fecha de creación de la imagen en la máquina host de Docker. Atención, no se trata de la fecha de creación de la imagen por el almacén, sino de la creación de la caché local.
- VIRTUAL SIZE (1.84kB): el tamaño de la imagen. Se trata de un tamaño virtual que representa a todas las capas que componen sucesivamente la imagen. El tamaño realmente ocupado en disco duro por una imagen, normalmente es muy inferior. En nuestro caso, la capa hello-world no pasa de 1,84 kilo-bytes y se trata además de su tamaño efectivo, porque no se basa en una capa inferior.

Este primer ejemplo de ejecución de un contenedor nos ha permitido mostrar los comandos básicos de Docker. A continuación, se va a avanzar un poco más en las operaciones de Docker, sin aventurarse por el momento en operaciones complejas.

# Utilizar imágenes Docker preexistentes

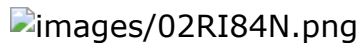
## 1. El Docker Store

### a. El principio

El primer ejemplo de ejecución de Docker se ha mostrado anteriormente, en el que se utiliza una imagen existente, porque se trata del medio más sencillo de establecer los contenedores. En la práctica, utilizar contenedores existentes en lugar de crear los suyos propios representa la mayoría de los casos de uso.

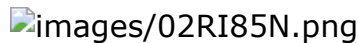
El éxito de Docker reside, no solamente en la herramienta en sí misma, sino también para una gran parte, en el hecho de que Docker - ayudado por una abundante comunidad - pone a disposición del público un gran número de imágenes pre-construidas. De esta manera es posible desarrollar contenedores con Apache, Nginx, PostgreSQL, MongoDB, WordPress y centenares de otros servidores y aplicaciones utilizando un sencillo comando, eventualmente acompañado de la configuración adecuada.

El sitio web Docker Store, accesible en <https://store.docker.com>, da acceso a estas imágenes:

images/02RI84N.png

### b. Búsqueda y cualificación de imágenes

Un clic en el enlace **Explore**, en la parte superior derecha de la interfaz, no permite acceder a una interfaz de búsqueda de imágenes (observe que no es imprescindible crear una cuenta para acceder a esta parte del sitio web):

images/02RI85N.png

La columna de la izquierda permite filtrar la búsqueda por categorías y por medio de otras opciones que vamos a detallar. También es posible filtrar por una expresión textual, al entrar en la zona de texto junto a la lupa, en la parte superior de la interfaz (aunque no sea visible en la captura de pantalla anterior).

La primera opción de filtro en la parte superior izquierda, merece una explicación particular. Por defecto, se posiciona en **Store**, lo que quiere decir que las imágenes que aparecerán han sido verificadas por Docker, lo que hace que tengan un mayor nivel de seguridad que las imágenes comunitarias potencialmente archivadas por cualquiera en el registro. La única condición para esta operación era crear una cuenta, no hay absolutamente ninguna garantía asociada y las imágenes pueden contener errores de seguridad intencionados, virus o cualquier tipo de ataque. Por lo tanto, hay un interés real cuando están disponibles, en utilizar imágenes Docker de tipo Store. Por el contrario, teniendo en cuenta su trabajo de validación, lógicamente no son más que algunos centenares, en comparación con los centenares de miles de imágenes disponibles en la sección Community. Además, como el nombre Docker Store indica, algunas son de pago o gratuitas pero en este último caso, con una licencia de utilización que pueda constituir otro tipo de freno al uso.

La segunda opción de filtro de las imágenes se corresponde con el carácter oficialmente certificado de las imágenes. En este caso, la verificación por Docker va incluso más lejos, con un compromiso de la empresa sobre el contenido de las imágenes y su rendimiento en una infraestructura certificada. En caso de la Store, el nivel de confianza viene de la relación entre Docker y los editores establecidos, así como de un eventual soporte de pago, lo que ya es un primer nivel de calidad interesante para las empresas, tradicionalmente cautelosas respecto a Docker por motivos del origen incierto de las imágenes del registro.

### c. Ejemplo de búsqueda

Supongamos que necesitamos establecer un servidor Nginx. Escribiendo esta palabra clave en la parte superior de la interfaz y validando la búsqueda, obtenemos lo siguiente:



Como se puede ver, el filtro todavía está en **Store**, lo que quiere decir que queremos ver imágenes validadas por Docker. Por el contrario, la ausencia del icono azul **Docker Certified**, muestra que estas imágenes no están certificadas. En la práctica, se muestra la segunda imagen porque se trata de un producto basado en Nginx, pero esto no es lo que buscamos. Por lo tanto, la primera es la imagen oficial que buscamos (como se indica en la etiqueta, aunque no fuera lo previsto).


El número de descargas es también un buen indicador de calidad de una imagen, si es significativamente elevado. Después de algunos millones de descargas (en nuestro ejemplo, más de diez millones), se puede considerar que una mala calidad (rendimiento, adaptación, funcionalidad, etc.) o una corrupción (virus, backdoor, bomba lógica, etc ) se hubiera detectado con toda seguridad.

Las etiquetas situadas debajo del nombre de la imagen dan información adicional de las plataformas destino y las categorías eventuales de pertenencia del resultado de la búsqueda. A continuación, un clic en el resultado de la búsqueda en sí misma conduce a la página correspondiente con, esta vez, todos los detalles asociados:

images/02RI87N.png

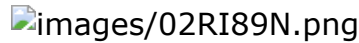
#### **d. Caso de las imágenes de la comunidad**

Volviendo una página atrás y cambiando el filtro para ver solo las imágenes generadas por la comunidad, en esta misma búsqueda con la palabra clave **nginx**, la gran cantidad de resultados muestra claramente el problema para un principiante:


images/02RI88N.png

Entre las 22.822 imágenes encontradas, algunas se corresponden con usos particulares de Nginx (como un proxy, controlador de tipo ingreso para Kubernetes, etc.), otras con instalaciones de Nginx que incluyen frameworks de desarrollos (Nginx con PHP y FPM, Nginx con Drupal, etc.); algunas son el resultado de empresas reconocidas por la calidad de sus imágenes, otras provienen de una persona que ha utilizado la parte pública de Docker Store para transmitir imágenes de una plataforma a otra sin voluntad real de compartir (documentación inexistente es un mal síntoma); algunas se han descargado millones de veces, otras casi nunca y son visiblemente obsoletas. En resumen, el efecto para una persona que busca simplemente una imagen con Nginx, puede ser el de una feria de imágenes sin ningún sentido, pero es necesario pasar por las imágenes de la comunidad si no hay disponible ninguna imagen oficial y crear una imagen por sí mismo no sea una opción.

En este caso, la calidad de la documentación es un primer indicio. Tomemos como ejemplo la imagen Nginx ofrecida por Bitnami.



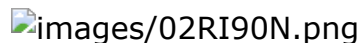
La presencia de etiquetas hace referencia a la integración continua CircleCI y la presencia de un chat dedicado, son síntomas de garantía. El contenido detallado de la sección **Overview** también es un buen signo para el potencial usuario. En la parte superior de la interfaz se encuentra el número de estrellas dadas por los usuarios a esta imagen (en nuestro ejemplo, la imagen bitnami/nginx tiene 36 estrellas, de ahí la visualización entre 30 y 40). Incluso aquí, se trata de un índice adicional de calidad. Independientemente, cada uno de estos índices solo tiene una fiabilidad limitada, pero tener índices positivos conduce al final a un nivel de garantía potencialmente suficiente, en todo caso para unas pruebas y una cualificación antes de entrar en producción.

 Es necesario estar identificado para votar y asignar una estrella a una imagen dada.

## e. Complementos a las imágenes oficiales

Contrariamente a las imágenes de la comunidad cuyo nombre utiliza el prefijo del identificador del propietario de la imagen (separado del nombre de la imagen por un símbolo /), las imágenes oficiales no usan prefijo. En el pasado, esto era equivalente a un prefijo de identificador "library", pero este ya no es el uso y la distinción entre imágenes oficiales y no oficiales a partir de ahora es más marcada.

Observe también que la lista de las imágenes oficiales ahora está en el almacén GitHub llamado `docker-library/official-images` (<https://github.com/docker-library/official-images>, y después desciende en el directorio llamado `library`):

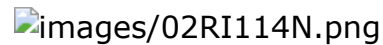


## f. Enlace con el registro Docker Hub

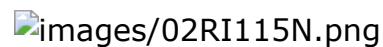
El Docker Store es una creación relativamente reciente, pero desde siempre existe la capacidad de basarse en las imágenes Docker ofrecidas por otros (editores o comunidad) y es una de las razones principales del éxito de Docker. En el pasado, la

publicación de imágenes era gestionada por el Docker Hub, algunas veces llamado registro Docker o registro público.

El Docker Hub existe siempre y es accesible en la dirección: <http://hub.docker.com>



Por supuesto, un clic en el enlace **Explore** en la parte superior derecha, redirige hacia una página donde claramente se indica en la parte superior que el nuevo lugar para encontrar el contenido Docker es el Docker Store, con un enlace que dirige hacia este último:



Claramente, el registro público Docker Hub sigue presente por razones de compatibilidad, pero parece una buena opción tener la costumbre de utilizar el Docker Store, que será más duradero.

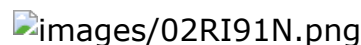
## g. Búsqueda por la línea de comandos

El cliente Docker también permite buscar las imágenes en función de palabras clave:

### Buscar una imagen Docker

```
docker search --filter=stars=20 nginx
```

La ejecución de este comando implica la búsqueda de todas las imágenes que contiene la palabra clave `nginx` y al menos dispongan de veinte estrellas. Esto conduce a la siguiente visualización o algo parecido:




Las columnas de información devuelven:

- El nombre de la imagen. Observe en particular que la primera imagen (con mejor nota, la ordenación se hace por el número de estrellas de manera descendente), está compuesta únicamente de la palabra `nginx`, sin prefijo correspondiente a un propietario. Como se ha explicado, se trata de la convención para llamar a las imágenes oficiales. Salvo necesidad particular, si queremos un contenedor con el servidor web Nginx, esta es la imagen que utilizaremos.



- La descripción de la imagen, que se alienta especialmente durante el registro de una imagen, de manera que los potenciales usuarios puedan conocer las particularidades de una imagen. Por defecto, la visualización está cortada, pero la opción `--no-trunc=true` permite ver las descripciones completas si es necesario.
- El número de estrellas.
- Una indicación del carácter oficial de la imagen.
- Una indicación del hecho de que la compilación de la imagen se ha realizado de manera automática.

 La antigua opción `--stars` está obsoleta y se sustituye por la opción `--filter`, que es más genérica. Observe también que esta última se puede sustituir por su versión abreviada `-f`, pero de manera general, las opciones se utilizan en el libro en su forma completa, lo que permite una mejor comprensión de su efecto.

Para encontrar las opciones, están disponibles los siguientes comandos:


#### Obtener la lista de los comandos del cliente Docker

```
docker help
```

#### Obtener ayuda de las opciones de un comando

```
docker help nombre_del_comando
```

Por supuesto, el contenido de la ayuda se limita a la explicación de las opciones del comando, pero para obtener todo el detalle de su contenido algunas veces es necesario hacer referencia a la documentación en línea. En el caso de la opción `--filter` del comando `docker search`, la URL <https://docs.docker.com/engine/reference/commandline/search/#filtering> contiene las explicaciones completas:

 `images/02RI92N.png`

## **h. Precauciones con una imagen no oficial**

Una imagen no oficial puede venir de cualquiera y contener cualquier funcionalidad. Es conveniente tener un mínimo de

seguridad durante su implantación. Por supuesto, la estanqueidad inherente al modo de funcionamiento de Docker, da una cierta garantía de ausencia de impactos en la máquina host, pero esto no impide comportamientos potencialmente peligrosos en el contenedor en sí mismo.


Tomemos como ejemplo la imagen `jpgougoux/mathapi`. Una búsqueda en el registro conduce a la página de bienvenida que describe rápidamente lo que se supone que se puede hacer con la imagen, cómo utilizarla y algunas propiedades adicionales.

images/02RI93N.png

El hecho de que la documentación indique un almacén GitHub como fuente de la imagen, es un primer paso hacia una validación del contenido de la imagen. En este caso, Docker ha creado la imagen a partir del código fuente disponible y en particular, del archivo Dockerfile. Esto permite, no únicamente validar que no se ha traficado con la imagen disponible por parte de la persona propietaria del almacén en Docker Store (es Docker el que se encarga de su compilación a partir de los archivos fuente), sino que además permite echar un vistazo al Dockerfile, que aparece justamente en la cuarta pestaña de la interfaz:

images/02RI94N.png

A continuación vamos a ver más en detalle el contenido de este archivo, pero de momento su presencia permite eliminar la opacidad sobre el contenido de la imagen.

 Esta opacidad no es en sí misma un problema desde el momento en que la confianza se establece con el proveedor. En las arquitecturas orientadas a servicios, es incluso lógico que la implementación sea lo más oculta posible y que solo sea visible el contrato de servicio. Por ejemplo, antes de analizar la imagen mencionada anteriormente, no conocíamos la tecnología subyacente y es perfectamente posible utilizarla, sin conocer el servidor de aplicaciones ni el lenguaje utilizados.

Si es necesario, todos los archivos se podrán controlar yendo al almacén origen en GitHub:

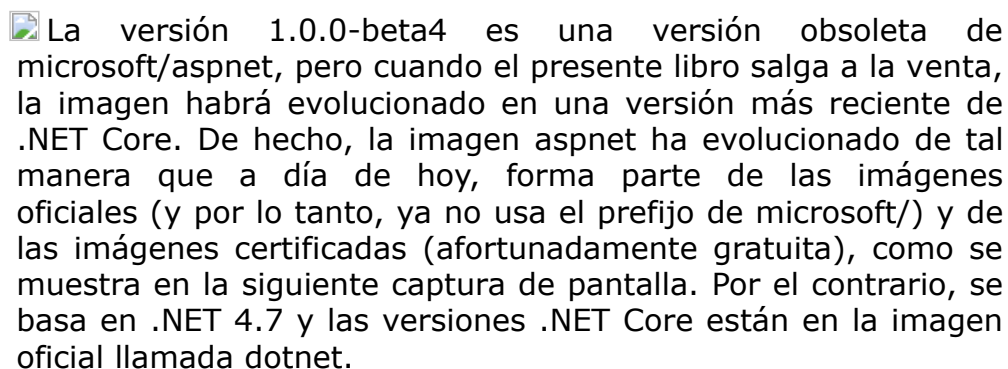
Images/102.png

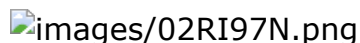
Por supuesto, el archivo `Dockerfile` es el mismo que el indicado en Docker Store:

Images/103.png

Para completar el análisis, conviene verificar su contenido, con un sentido crítico, incluso la asistencia de un profesional si el uso previsto es para producción.


En primer lugar, la palabra clave `FROM` hace referencia a una imagen que, aunque no sea oficial, proviene de un origen más reconocido, a saber, la cuenta llamada `microsoft`. Atención, el hecho de tener una cuenta con nombre, no quiere decir que la persona represente a la empresa asociada. Docker no tiene ningún interés en dejar que las personas usurpen las cuentas de empresas conocidas. En el caso particular de esta imagen `microsoft/aspnet`, un vistazo a la página del registro permite encontrar un enlace al sitio web MSDN oficial de Microsoft y que reenvía a la página del registro Docker. Por tanto, se establece la relación de confianza.

La versión `1.0.0-beta4` es una versión obsoleta de `microsoft/aspnet`, pero cuando el presente libro salga a la venta, la imagen habrá evolucionado en una versión más reciente de `.NET Core`. De hecho, la imagen `aspnet` ha evolucionado de tal manera que a día de hoy, forma parte de las imágenes oficiales (y por lo tanto, ya no usa el prefijo de `microsoft/`) y de las imágenes certificadas (afortunadamente gratuita), como se muestra en la siguiente captura de pantalla. Por el contrario, se basa en `.NET 4.7` y las versiones `.NET Core` están en la imagen oficial llamada `dotnet`.

images/02RI97N.png

Si este no es el caso, la práctica consiste en seguir el origen de las imágenes las unas respecto a las otras. En casi todos los casos, en algún momento nos encontraremos necesariamente con una imagen que provenga de un proveedor para el que, la confianza es lo suficientemente fuerte como para que no tengamos necesidad de su contenido. Falta comprobar todos los comandos adicionales que están en los archivos `Dockerfile` sucesivos, de manera que se valide que no añaden funcionalidades maliciosas.

En algunos casos es relativamente sencillo validar el contenido. Por ejemplo, si volvemos a la imagen `jpgouigoux/mathapi`, hay tres archivos guardados y su contenido es fácilmente verificable en el mismo almacén GitHub que el archivo `Dockerfile`. Un vistazo al archivo `Service.cs`, por ejemplo, muestra claramente que el contenido es el esperado.

 La respuesta a la observación de que hace cierto tiempo es, para los más curiosos, que este servicio se implementa en ASP.NET 5 / MVC 6, en el que se utiliza el framework .NET Core, a saber, la versión ligera y moderna de .NET, soportada por Microsoft en Windows, pero también en Linux. Una parte de esta información estaba escrita en la página de bienvenida, pero un vistazo más en profundidad permite comprobar su veracidad.

En otros casos, el análisis es más complejo. Por ejemplo, si no se hubiera establecido que la imagen `aspnet` era de confianza, hubiera sido necesario analizar el contenido del `Dockerfile`, representado a continuación:

```
FROM microsoft/dotnet-framework:4.7

SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

RUN Add-WindowsFeature Web-Server; \
    Add-WindowsFeature NET-Framework-45-ASPNET; \
    Add-WindowsFeature Web-Asp-Net45; \
    Remove-Item -Recurse C:\inetpub\wwwroot\*

ADD ServiceMonitor.exe /

#download Roslyn nupkg and ngen the compiler binaries
RUN Invoke-WebRequest
https://api.nuget.org/packages/microsoft.net.compilers.2.3.1.nupkg
-OutFile c:\microsoft.net.compilers.2.3.1.zip; \
    Expand-Archive -Path c:\microsoft.net.compilers.2.3.1.zip
-DestinationPath c:\RoslynCompilers; \
    Remove-Item c:\microsoft.net.compilers.2.3.1.zip -Force; \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\csc.exe
/ExeConfig:c:\RoslynCompilers\tools\csc.exe | \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\vbc.exe
/ExeConfig:c:\RoslynCompilers\tools\vbc.exe | \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\VBCSCompiler.exe
/ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe | \
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\csc.exe
/ExeConfig:c:\RoslynCompilers\tools\csc.exe | \
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\vbc.exe
```

```
/ExeConfig:c:\RoslynCompilers\tools\vbc.exe | \  
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe  
install c:\RoslynCompilers\tools\VBCSCompiler.exe  
/ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe;  
  
ENV ROSLYN_COMPILER_LOCATION c:\\RoslynCompilers\\tools  
  
EXPOSE 80  
  
ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
```

La imagen básica es oficial, el archivo es mucho más completo que el de la imagen estudiada antes. Sobre todo, descargar y descomprimir un archivo cuyo contenido es opaco, se debería estudiar con cuidado para eliminar cualquier duda. Este análisis hace necesario disponer de una experiencia avanzada para validar que el contenido no tiene efectos indeseados. En ausencia de una competencia como esta, la pregunta vuelve de nuevo una vez más a la confianza en el productor de la imagen. Esta es la razón por la que Docker orienta cada vez más al usuario a las imágenes oficiales.


## 2. Gestión de la cuenta Docker Hub y almacenes privados

Hasta ahora, solo hemos consumido imágenes Docker desde el registro, y estas imágenes eran públicas. Por lo tanto, no hay necesidad de conectarse al Docker Store. Sin embargo, con el objetivo de acceder a las imágenes en el registro público, es necesario tener una cuenta activa y conectarse. Esta identificación también sirve para acceder a los almacenes privados. Como su nombre indica, los almacenes privados son almacenes de imágenes Docker, a los que solo pueden acceder las personas con autorización para explotarlas.

Docker ofrece cuentas gratuitas o de pago. La limitación de las primeras es que solo permiten un único almacén privado. El número de almacenes públicos es ilimitado (en la actualidad).

### a. Creación de una cuenta

El procedimiento de creación de una cuenta no necesita explicaciones. Es suficiente con conectarse a <https://hub.docker.com> y hacer clic en el comando **Login**, para acceder a la ventana de conexión, en la que el enlace **Create Account** permite solicitar una cuenta:

 images/02RI98N.png

La siguiente página no pide nada especial:



Una vez visitado el enlace de confirmación que se envía al correo electrónico, la cuenta está lista para la conexión y la imagen de su perfil aparece en la parte superior derecha de la interfaz y confirma que está conectado.

## b. Características de la cuenta


La cuenta que se acaba de crear es una cuenta gratuita. No está limitada respecto al número de almacenes de imágenes públicos, pero solo permite un único almacén privado. Preste atención al sentido de la palabra almacén; en el lenguaje Docker, se trata de un conjunto de imágenes con el mismo nombre, pero potencialmente las etiquetas son diferentes.

Otro límite es que solo se puede lanzar a la vez una operación de compilación de una imagen, desde su código fuente en GitHub (ver un poco más adelante los detalles). Por necesidades de las pruebas o de los desarrollos, esto en general es suficiente.

Si estas condiciones son demasiado restrictivas para su uso, es posible recurrir a ofertas de pago, que le darán acceso a más almacenes privados y capacidad de compilaciones en paralelo (el vocablo inglés es *build*). La dirección <https://hub.docker.com/billing-plans> lista las diferentes ofertas existentes a día de hoy:



Un clic en el botón **Upgrade Plan**, correspondiente a la oferta elegida, conduce a una interfaz de elección del medio de pago. A continuación, la oferta puede evolucionar en función de las necesidades.

 En función de sus necesidades, puede ser pertinente establecer su propio registro privado o bien echar un vistazo a las ofertas comerciales como Azure Container Registry. Un capítulo del presente libro se dedica a la gestión de los registros.

## c. Automated build y cuenta GitHub

Como se ha explicado anteriormente es posible hacer que Docker Hub sea responsable de la compilación de una imagen Docker, a partir del código fuente depositado en GitHub. Este modo de funcionamiento marcado como Automated Build en los almacenes, permite dar una garantía al usuario final de que la imagen en Docker Hub se corresponde con el archivo Dockerfile mostrado, así como permitir al desarrollador deshacerse completamente de esta operación, que se realizará automáticamente en cada modificación del almacén del código fuente.

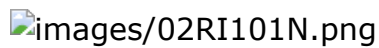
Para esto, es necesario que las cuentas Docker Hub y GitHub estén relacionadas. Este último punto merece una explicación un poco más detallada que para la creación de una cuenta.

Conéctese con la cuenta creada en Docker Hub:  
<http://hub.docker.com>


Pulse en el nombre de su cuenta, en la parte superior derecha.

Seleccione el comando **Settings**.

Vaya a la pestaña **Linked Accounts & Services**.

images/02RI101N.png

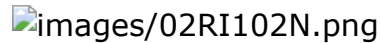
Como se ha explicado en la página anterior, el enlace con una cuenta de tipo GitHub o Bitbucket le permite establecer un sistema de build automático de sus imágenes. Si ha puesto en marcha un almacén GitHub que contiene un archivo `Dockerfile` (de manera tradicional en la raíz, incluso si es posible funcionar de otra manera), es posible utilizar un webhook para que cada modificación añadida al almacén GitHub lance la compilación de la imagen Docker y su puesta a disposición en la nueva versión en Docker Hub.

 Un webhook es un tipo de evento transmitido entre dos aplicaciones web. Se caracteriza por una llamada de URL realizada por el emisor, acompañada de argumentos que dependen del contexto. El receptor puede desencadenar una operación particular en función de estos valores. La tecnología de las webhooks no se especifica en Docker Hub o GitHub. Se trata de una práctica habitual en los servicios web.

Vamos a establecer esta solución en la solución `mathapi`, utilizada como el ejemplo mostrado anteriormente.

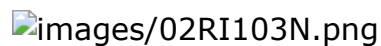
Pulse en **Link Github** en el icono GitHub.

En la pantalla siguiente, seleccione la opción de acoplamiento recomendada. Esta opción da más permisos a Docker Hub para manipular su cuenta GitHub y precisamente, establecer automáticamente las webhooks para usted.



Si no está conectado a GitHub, es necesario realizar esta autenticación.

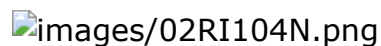
En la pantalla que sigue, valide la autorización dada sobre los almacenes GitHub que desee:



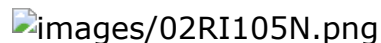
Es posible que tenga que introducir su contraseña GitHub de nuevo, para confirmar esta autorización.

De vuelta a su cuenta Docker Hub, pulse en el botón **Create** y después en **Create Automated Build**.

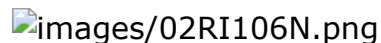
Seleccione el tipo de fuente que desea, en nuestro ejemplo **GitHub**.



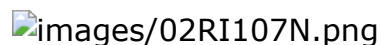
La lista de sus almacenes GitHub se muestra, agrupados por organización. Seleccione el almacén de origen a compilar.



Se muestra una ventana con varias opciones.



Pulse en **Click here to customize behavior** para que aparezcan las opciones de compilación:



Es posible lanzar la build automatizada cuando se modifica una rama o por supuesto, una etiqueta (Tag). La segunda columna permite especificar el nombre de la rama o de la etiqueta en cuestión. La tercera se corresponde con la localización del archivo `Dockerfile` que hay que utilizar. Habitualmente, está en la raíz del almacén GitHub pero si no es el caso, esta opción permite adaptar el comportamiento de la build automática. Para terminar, la cuarta columna en las líneas de opción permite especificar la




etiqueta que se adjuntará a la imagen Docker, una vez compilada y ubicada en el almacén Docker Hub.


Como no utilizaremos estas opciones, puede pulsar en **Revert to default settings**.

Después de haber introducido una descripción (es un campo obligatorio), pulse en **Create**.

La interfaz apunta entonces al almacén creado junto con el código fuente y es posible encontrar los argumentos de lo que se acaba de configurar en la pestaña **Build Settings**.

 Images/02RI108N.png

El botón **Trigger** permite lanzar manualmente una build de la imagen a partir del código fuente. También hay un enlace a los resultados de la build, en la pestaña **Build Details**. Esto es práctico para solucionar posibles problemas. También podemos comprobar que el almacén Docker Hub ha recuperado y utilizado el `Readme.md` de GitHub y que una pestaña muestra el `Dockerfile` utilizado, lo que es mucho más limpio que dirigir simplemente por medio de un enlace al almacén GitHub, como se había implementado en el primer enfoque:

 Images/02RI109N.png

Esta manera de proceder con un almacén de código fuente, relacionado por medio de un evento al almacén de imagen por un webhook, es el estándar de facto de producción de imágenes Docker, y se corresponde con la filosofía de separación correcta de las responsabilidades que impera en los servicios web y más allá, en cualquier sistema informático correctamente urbanizado. Cada uno debe hacer su trabajo y solo su trabajo, de manera que se haga lo mejor posible.

 Para más detalles de la configuración avanzada de las builds automatizadas, hay una documentación muy completa disponible en <http://docs.docker.com/docker-hub/builds>.

#### d. Conexión a la cuenta en línea de comandos


El modo de operación de las imágenes en el registro, sobre un navegador, se presta a la búsqueda de imágenes por palabras clave o a su administración, pero cuando se trata de poner en producción, es necesario tomar el control de estas imágenes e

instalarlas de manera efectiva. Aunque existen interfaces gráficas para Docker, en la práctica se utiliza la línea de comandos. Con el objetivo de acceder a los almacenes privados, es necesario poder conectarse a su cuenta Docker Hub desde la línea de comandos.

### Conexión a una cuenta Docker Hub

```
docker login
```


La conexión solicita indicar el nombre de la cuenta creada en Docker Hub, la contraseña, así como una dirección de correo electrónico:

 images/02RI110N.png

En el siguiente capítulo volveremos sobre la herramienta de la cuenta Docker, para poder añadir imágenes compiladas localmente en el registro, de manera pública o privada. Por el momento, terminamos esta sección con el comando inverso del anterior:

### Desconexión de la cuenta Docker Hub

```
docker logout
```

 Es importante observar que es posible utilizar Docker sin tener cuenta en DockerHub. En efecto, hay numerosas ventajas que se pueden extraer de las imágenes existentes. Incluso cuando un usuario crea sus propias imágenes, perfectamente puede pasarlas de una máquina a otra sin recurrir al alojamiento ofrecido por Docker. La primera solución es exportar las imágenes en los archivos. La segunda es establecer su propio almacén de imágenes. Estudiaremos estas dos posibilidades en el siguiente capítulo.

## **e. Webhook en evento de push en Docker Hub**

Más atrás hemos visto el uso de las webhooks de GitHub para Docker Hub, para lanzar la compilación automática de una imagen durante una operación de modificación de una rama o etiqueta, en un almacén GitHub dado. También existe una funcionalidad de webhook en la que Docker Hub es el emisor.

Las páginas de la documentación Docker correspondientes a esta funcionalidad (<https://docs.docker.com/docker-hub/repos/#webhooks> y <https://docs.docker.com/docker->


[hub/webhooks/](#)), ofrecen un ejemplo del contenido JSON (*JavaScript Object Notation*), que se enviará a un callback durante un evento de push:

```
{
  "callback_url":
  "https://registry.hub.docker.com/u/svendowideit/busybox/hook/2141bc0c
  dec4hebec411i4c1g40242eg110020/",
  "push_data": {
    "images": [

      "27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3"
    ,
      "51a9c7c1f8bb2fa19bcd09789a34e63f35abb80044bc10196e304f6634cc58
      2c",
      "...
    ],
    "pushed_at": 1.417566822e+09,
    "pusher": "svendowideit"
  },
  "repository": {
    "comment_count": 0,
    "date_created": 1.417566665e+09,
    "description": "",
    "full_description": "webhook triggered from a 'docker push'",
    "is_official": false,
    "is_private": false,
    "is_trusted": false,
    "name": "busybox",
    "namespace": "svendowideit",
    "owner": "svendowideit",
    "repo_name": "svendowideit/busybox",
    "repo_url":
    "https://registry.hub.docker.com/u/svendowideit/busybox/",
    "star_count": 0,
    "status": "Active"
  }
}
```

El callback solo se llama si la operación de push se desarrolla correctamente.

En la actualidad, el único evento desencadenado por Docker Hub es el que se corresponde con esta operación de push. Este evento se repite en cada operación de almacenamiento, incluso si la imagen ubicada no ha cambiado. Además, se activa en un almacén manual, pero también durante un almacenamiento como consecuencia de una compilación automática. La configuración de un webhook se realiza en la pestaña del mismo nombre en un almacén Docker Hub.

images/02RI111N.png

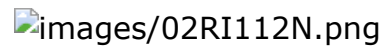
En un almacén estándar, también es posible encadenar las webhooks, con una ejecución en sucesión (la llamada a la

siguiente callback en una lista encadenada, solo se realizará cuando la anterior se haya validado, devolviendo correctamente la URL llamada), pero esto va más allá del marco del presente libro.

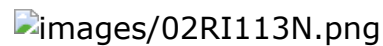
## f. Desconexión de las cuentas Docker Hub y GitHub


Si por casualidad desea desconectar las cuentas Docker Hub y GitHub, el primer reflejo sería realizar la operación inversa a la mostrada más arriba, accediendo a su perfil Docker Hub y sus argumentos.

En la pestaña **Linked Accounts & Services**, sería posible eliminar el enlace a GitHub:



Sin embargo, esto no es suficiente para poner la situación en su estado inicial, porque también haría falta eliminar la autorización de Docker Hub en los argumentos de GitHub, haciendo clic en el enlace **Revoke** asociado en la lista **Authorized OAuth Apps**:



 Preste atención con eliminar el enlace en primer lugar. Si elimina la autorización en primer lugar, el icono de eliminación de enlace ya no se podría pulsar y sería necesario desconectarse y volver a conectarse a su cuenta Docker Hub para poder desconectar la cuenta.

## Un segundo contenedor

Después de esta primera ejecución de un contenedor de tipo `hello-world`, así como de las disquisiciones sobre la manera de encontrar las imágenes ya existentes en el registro Docker Hub, vamos a pasar a una segunda etapa que consiste en lanzar un contenedor un poco más complicado, con la que esta vez vamos a ver una interacción real. La imagen `hello-world` se contenta con mostrar un texto. Esto ha permitido abordar sin problemas lo que sucede en el fondo, pero es evidente que una sencilla visualización como esta no es muy útil.

Nuestra segunda operación consistirá en escribir archivos en un contenedor de tipo `ubuntu`. La imagen `ubuntu`, como su nombre indica, contiene simplemente una instalación minimalista del sistema operativo. El hecho de que el nombre no esté prefijado, significa que se trata de una imagen oficial.

### 1. Recuperación de la imagen


Vamos a comenzar simplemente descargando la imagen, lo que implicaría abordar la noción de tag.

#### Recuperación de una imagen en su última versión

```
docker pull ubuntu:latest
```

La descarga llevaría algunos segundos, incluso minutos, en función de su conexión a Internet. Sin embargo, esta carga de la imagen solo tiene lugar una única vez y cualquier ejecución posterior se basará en esta imagen, incluidas además (lo que representa una de las grandes fortalezas de la arquitectura por capas) todas las imágenes que se basan en esta (y son numerosas, Ubuntu siendo una distribución básica frecuentemente utilizada para las imágenes Docker).

El cliente Docker devuelve información de la descarga actual:

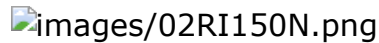
 images/02RI149N.png

Una vez que se realiza esta descarga, un comando permite ver las imágenes disponibles localmente.

## Visualizar las imágenes Docker presentes en la máquina

```
docker images
```

El resultado sería una visualización como se muestra a continuación:

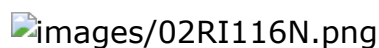


## 2. Explicación de los tags

Vemos que el tag asociado a la imagen `ubuntu` es `latest`, como se había indicado en el comando `pull`. El tag es un tipo de marcador de versiones de la imagen. Sin embargo, al contrario de lo que sucede con un sistema de versiones, es posible tener varios tags para una misma versión. Por ejemplo, la última versión que hemos descargado está marcada con el tag `latest`, para caracterizar este hecho, pero también está marcada con el tag `16.04`, porque se trata de esta versión de Ubuntu. También se conoce con el tag de `xenial` (el nombre de release de la versión 16.04 es *Xenial Xerus*). En el pasado, tenía una etiqueta con una versión de tres cifras, pero se sigue la lógica de la Semantic Versioning, una persona querría utilizar una versión 16.04 tomando necesariamente la última en fecha. Por el contrario, las etiquetas de las últimas compilaciones existentes (en nuestro ejemplo existe un tag `xenial-20170915`), se corresponden con Xenial Xerus del 15 septiembre del 2017.

Observe que si simplemente hemos indicado `ubuntu` por medio del comando `pull`, automáticamente habríamos recuperado la etiqueta `latest`. El resultado habría sido el mismo que el del comando anterior.


Por el contrario, es interesante ver que si el comando utilizado es `docker pull ubuntu:xenial-20170915`, la descarga es inmediata: la imagen es la misma, con dos etiquetas que apuntan al mismo contenido. La visualización de la caché de registro local confirma esto, los identificadores hexadecimales son completamente idénticos:



Encontramos los dos tags asociados a la misma imagen (la columna `IMAGE ID` muestra cada vez el mismo valor

2d696327ab2e, que es el identificador único de la imagen, lo que demuestra que se trata de la misma imagen para todos los tags).

La lista devuelve también la fecha de creación de la imagen (no la fecha de creación de la copia en la cache de la máquina host, sino la fecha de build de esta máquina por la organización encargada de su mantenimiento), así como su tamaño virtual. El tamaño virtual, en este caso particular, se corresponde con el tamaño de la capa descargada, porque la imagen `ubuntu` no se basa en otra imagen preexistente. Es fácil comprobar esto mirando el `Dockerfile` en [https://registry.hub.docker.com/\\_/ubuntu/](https://registry.hub.docker.com/_/ubuntu/), que muestra que el valor de `FROM` es `scratch` (el código para indicar ninguna imagen).


 Es posible que el lector se pregunte por qué el tag `latest` no está asociado a la versión 17.10 "Artful" o incluso a la 17.04 "Zesty". Es una elección de Canonical (el editor de Ubuntu que también mantiene la imagen Docker), asociar el tag `latest` a la última versión cuyo soporte es extendido. La última versión Long Term Support es la 16.04, de ahí la asignación del tag `latest` a esta última, incluso si hay otras versiones más recientes.

### 3. Primera ejecución

Se puede realizar una primera ejecución sin ninguna opción, con la siguiente línea de comandos.

#### Ejecución sin opción de un contenedor

```
docker run ubuntu
```

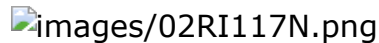
 También se hubiera podido especificar un tag detrás del nombre de la imagen a ejecutar. Por defecto, el comando `run` utilizará el tag `latest` y el anterior comando es el exacto equivalente de `docker run ubuntu:latest`.

Desde un punto de vista del operador hay que esperar un poco, y después Docker devuelve el control. Parece como si nada hubiera sucedido... Por lo tanto, miremos si hay un contenedor activo:

#### Listar los contenedores activos

```
docker ps
```

La lista está vacía:

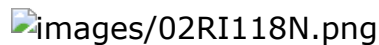


Para diagnosticar mejor lo que ha pasado, vamos a llamar al comando `ps` con la opción que permite listar todos los contenedores incluidos aquellos que ya no están más activos:

### Listar todos los contenedores, incluidos los inactivos

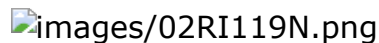
```
docker ps -a
```

Esta vez, en la lista vemos un contenedor lanzado y al cabo de un minuto aproximadamente:



Es el que se acaba de ejecutar y la columna `STATUS` indica efectivamente que el proceso se ha detenido de manera normal (código de salida 0) y al cabo de un minuto aproximadamente, por lo tanto, casi inmediatamente después de su ejecución. La columna `COMMAND` nos informa del proceso que se había ejecutado, a saber, un shell Bash.

Si echamos un vistazo al `Dockerfile` utilizado para construir nuestra imagen `ubuntu`, veremos en efecto que el proceso lanzado por defecto es `/bin/bash`.



¿Qué sucede si lanzamos un proceso de tipo shell sin asociarle una consola para pasarle órdenes? Simplemente se detiene todo, considerando que ha realizado el trabajo que se le ha dado, es decir, en este ejemplo concreto, nada. Esto es lo que pasa en nuestra primera ejecución de contenedor.

Por lo tanto, este primer contenedor no sirve para nada. Vamos a eliminarlo simplemente utilizando la etiqueta que se le había asignado (también podríamos utilizar el identificador o una parte significativa de este último).

### Eliminar un contenedor detenido

```
docker rm objective_swanson
```



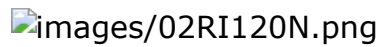
## 4. Ejecución en modo interactivo

Para poder realmente explotar el contenedor `ubuntu` y en particular actuar sobre el proceso de shell `Bash` que se lanza, vamos a ejecutar el contenedor en modo interactivo, es decir, asignándole una consola `TTY` para intercambiar y abrir la consola para que recibe las órdenes que entran en esta consola.

### Lanzar un contenedor en modo interactivo

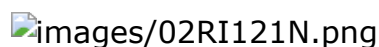
```
docker run -i -t ubuntu
```

De nuevo una vez más, Docker devuelve el control de manera casi inmediata, pero esta vez, la línea de comandos es diferente:




Ya no estamos identificados como el usuario actual de la máquina `host`, sino como `root` en la máquina llamada `a873185e03fc` o cualquier otro identificador de contenedor que Docker haya gestionado. Este identificador se utiliza a la vez para designar al contenedor y asignarle un nombre a la máquina que no podemos calificar de virtual, pero que está bien y correctamente simulado por el cambio del espacio de nombres.

Como en una máquina que sería efectivamente distinta, podemos ejecutar cualquier comando por el shell, por ejemplo `ls`, para ver los archivos presentes:



Como se trata de un contenedor estanco respecto a la máquina `host` y tampoco puede afectar a la imagen a partir de la que se ha creado, podemos autorizar cualquier comando que, sobre una verdadera máquina, sería potencialmente catastrófico, como por ejemplo, eliminar todo el directorio `/home`:



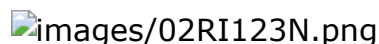
 Preste atención y asegúrese perfectamente de estar correctamente ubicado en el contenedor para, sobre todo, no ejecutar un comando tan peligroso en su máquina `host`. El prompt de la línea de comandos le informa sobre esto, mostrando el nombre de la máquina.

Una vez terminado el paquete, volvemos a la máquina host escribiendo el comando `exit` (que sale de la shell actual y por tanto del contenedor, porque este solo tiene por objetivo que este único proceso se ejecute). Podemos observar desde el exterior el daño causado, utilizando el comando adecuado.

### Encontrar la lista de modificaciones de un contenedor respecto a su imagen de ejecución

```
docker diff [identificador del contenedor]
```

En nuestro ejemplo, obtenemos la siguiente información:



images/02RI123N.png

La sintaxis utilizada es la de diff/patch en Linux, donde se utiliza `D` para las porciones eliminadas (aquí, el directorio `/home`) y `C` para las creadas (en nuestro caso, el directorio `/root` así como el archivo `.bash_history` creado automáticamente en este último, para almacenar el histórico de las acciones realizadas en el shell, que es un comportamiento estándar de Linux). Respecto a `A`, se corresponde con la adición de un archivo.

Afortunadamente, por el momento esta operación no se escribe en la capa agregada por Docker para gestionar las modificaciones del sistema de archivos, y no tiene ningún impacto en la capa inferior, a saber, la imagen `ubuntu` que, hagamos lo que hagamos, permanecerá como antes. Además podemos validarlo ejecutando el comando `docker run -i -t ubuntu` y comprobar que el directorio `/home` está presente:




images/02RI124N.png

Como muestra el cambio de nombre de la máquina en la línea de comandos, estamos en un segundo contenedor. Podríamos decir que `/home` está presente "de nuevo", pero esto no reflejaría realmente la realidad. Si ahora encontramos los dos contenedores que se han ejecutado, podríamos decir que en el primero, `/home` ya no existe. Pero en ausencia de modificaciones, está presente en el segundo porque se ha instanciado desde la misma imagen, que está y estará invariante.

## 5. Persistencia de las modificaciones en forma de imagen

Acabamos de ver que una imagen nunca se modificará por lo que podamos hacer dentro de un contenedor. Como se ha explicado previamente a nivel teórico, es una capa adicional la que recibe todas las modificaciones, y estas no se añaden a la imagen a partir de la que ha lanzado el contenedor. Pero supongamos ahora que deseamos persistir estas modificaciones más allá del ciclo de vida del contenedor, incluso cuando este se haya destruido.


Para esto, existe un comando que va a transformar la capa superior (que se había establecido por encima de las capas en modo solo lectura, para recibir las modificaciones) en una nueva imagen basada en las capas inferiores y que podrá servir para crear a su vez otros contenedores. De manera esquemática:

 Images/esquema14.png


### Persistir el estado de un contenedor en una nueva imagen

```
docker commit [identificador o nombre del contenedor] [nombre de la imagen]
```


En nuestro caso:

 images/02RI125N.png

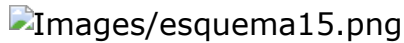
La imagen creada se comporta exactamente como cualquier otra imagen Docker. La única diferencia de jpgougoux/ubuntu\_nohome con Ubuntu, es que no se trata de una imagen oficial y que (todavía) no se ha archivado en el registro Docker Hub, disponible para el resto de personas.

 El tercer capítulo está dedicado a la creación de las imágenes en detalle y en particular muestra cómo cargarla en el almacén Docker Hub, para una compartición privada o pública.

Esto quiere decir en particular que a su vez es posible lanzar los contenedores basados en esta imagen:

 images/02RI126N.png

Si nos centramos en el esquema las dos capas que compone la nueva imagen en una sola, tendríamos una representación como la siguiente:



Por supuesto una vez instanciada esta imagen en un contenedor, se le va a asociar con una capa superpuesta para contener las modificaciones y así sucesivamente. Este enfoque por capas sucesivas es una buena práctica, fundamental en la explotación de Docker. Veremos las múltiples ventajas en términos de gestión y rendimiento que añade una separación correctamente realizada.

Si no desea conservar la imagen, hay un comando que permite eliminarla.

### Eliminar una imagen

```
docker rmi [nombre de la imagen]
```

## 6. Primeros pasos con el cliente Docker

Sin entrar en el detalle de todas las opciones ni en los aspectos complejos que vamos a tratar a continuación, vamos a detenernos en algunas opciones sencillas y útiles del comando `run`.

### **a. Limpieza en los contenedores**

Si repite varias veces las operaciones utilizadas como ejemplo para entender la utilización correcta de Docker, habrá un momento en el que querrá partir de cero y eliminar todos los contenedores actuales. En lugar de llamar al comando `ps` y después varias veces al comando `rm` en varios contenedores sucesivos, es posible acoplar los resultados de la primera en la ejecución de la segunda, de manera que se eliminen todos los contenedores de golpe.

Evidentemente, este tipo de operaciones solo tiene sentido en una máquina de desarrollo, pruebas o aprendizaje y, en ningún caso, en un servidor de despliegue de contenedores en producción.

Para que este comando único funcione, es necesario utilizar la opción `-q` del comando `ps`, de manera que solo muestre los identificadores de los contenedores y ninguna otra información. Estos son los datos que se pasarán al comando `rm`, que eliminará todos los contenedores con los identificadores proporcionados.

### Eliminar todos los contenedores

```
docker rm `docker ps -a -q`
```

Observe que las opciones se pueden unir y que por lo tanto, es posible en nuestro caso sustituir `docker ps -a -q` por `docker ps -aq`.

## **b. Limpieza en las imágenes**

El tiempo de descarga, así como el consumo de ancho de banda y de electricidad, hacen que este tipo de limpieza no sea una buena opción para las imágenes. Sin embargo, conviene señalar que es posible eliminar varias imágenes de golpe, llamando simplemente al comando dedicado con varios argumentos.

### Eliminar varias imágenes

```
docker rmi [imagen 1] [imagen 2] ... [imagen n]
```

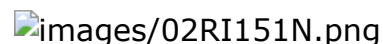
## **c. La gran limpieza**

Docker dispone desde versiones más recientes de un comando `system` y más concretamente de un sub-comando `prune`, que permite realizar la limpieza en los contenedores detenidos, las imágenes huérfanas y otros recursos a priori no utilizados.

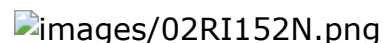
### Hacer limpieza en los objetos Docker

```
docker system prune
```

La visualización proporcionada por el comando, es la siguiente:

images/02RI151N.png

Algunas veces, la ejecución de este comando puede llevar mucho tiempo, pero es el precio a pagar para obtener una limpieza profunda. Al final de la ejecución, el comando indica el espacio recuperado:

images/02RI152N.png

## **d. Eliminación automática a la salida**

La velocidad de instanciación de los contenedores hace posible su utilización para tareas muy breves, donde la pesadez de las máquinas virtuales nos obligaría a ahorrar el número de instancias. El caso de estudio es el desarrollador que lanza pruebas unitarias y que, en lugar de lanzar un script de reinicialización de una base de datos, simplemente va a relanzar un contenedor.

En lugar de tener que hacer regularmente la limpieza de numerosos contenedores durante el ciclo de vida limitado de una ejecución, es más sencillo indicar desde el inicio que el contenedor se debe eliminar cuando realice su tarea, gracias a la opción `rm`.

### Lanzar y eliminar un contenedor desde el inicio del proceso

```
docker run --rm hello-world
```


## **e. Asignación de un nombre de contenedor**

Otro truco para facilitar la gestión de los contenedores, es asociarle nombres significativos durante la ejecución, de manera que no se tenga que hacer referencia a la salida del comando `ps` para conocer el nombre que se le haya asignado automáticamente. Para esto sirve la opción `--name` del comando `run`.

### Lanzar un contenedor con un nombre dado

```
docker run --name=[nombre asignado] [imagen a instanciar]
```

Este enfoque permite eliminar más fácilmente. El recurso del comando `ps` no es útil (se muestra a continuación solo para comprobar que se ha asignado correctamente el nombre).

 Images/02RI127N.png

## **f. Modificación del punto de entrada por defecto**

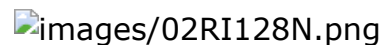
Anteriormente hemos visto que el archivo `Dockerfile` de la imagen `ubuntu`, contenía una instrucción `CMD` que apuntaba a `/bin/bash`. Esto hace que cuando se instancia un contenedor en esta imagen, se lance el proceso correspondiente a un shell `Bash`. Sin embargo, es posible sobrecargar este funcionamiento por

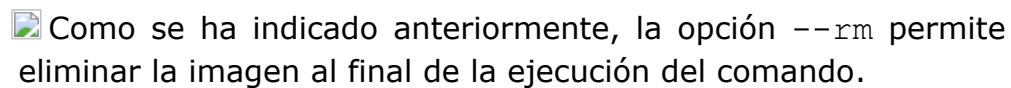
defecto y solicitar al comando `run` que lance otro comando en el contenedor una vez arrancado.

### Lanzar un contenedor con un comando dado

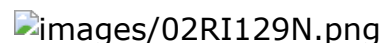
```
docker run [imagen a instanciar] [comando a ejecutar]
```

En nuestro caso, por ejemplo podríamos llamar al comando `env` para listar las variables de entorno presentes en la imagen `ubuntu`:

images/02RI128N.png

 Como se ha indicado anteriormente, la opción `--rm` permite eliminar la imagen al final de la ejecución del comando.

Para estar seguro de que son las variables de entorno del contenedor y no las de la máquina host, podemos ejecutar este comando antes del final de la ejecución del contenedor y comprobar que los resultados son muy diferentes:

images/02RI129N.png

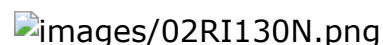
## **g. Enviar variables de entorno**

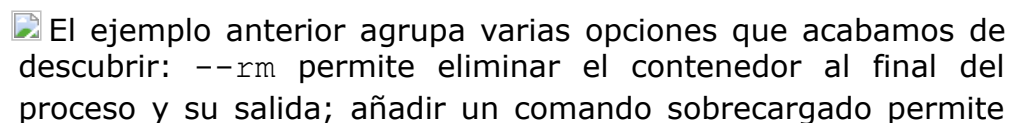
Mientras estamos con las variables de entorno, las opciones existen para inyectar estas últimas al contenedor lanzado. En primer lugar, es posible inyectar una variable de entorno existente en la máquina host, especificando simplemente su nombre.

### Inyectar una variable de entorno durante la puesta en marcha de un contenedor

```
docker run -e=[variable de entorno] [imagen a instanciar]
```

Por ejemplo, podríamos utilizar la variable `LOGNAME` en un contenedor:

images/02RI130N.png

 El ejemplo anterior agrupa varias opciones que acabamos de descubrir: `--rm` permite eliminar el contenedor al final del proceso y su salida; añadir un comando sobrecargado permite

ejecutar el proceso deseado sobrecargando el indicado por defecto en la imagen; para terminar, aquí se muestra que es posible pasar una variable de entorno del host al contenedor. Al final, creamos un contenedor pasándole una variable de entorno, solicitando que ejecute el comando `env` que muestra todos estos e indicando que se debe destruir una vez que el comando se haya ejecutado. Todo esto se indica en una única línea de comandos.

También es posible crear una variable de entorno que no existe en el host.

#### *Añadir una variable de entorno durante la puesta en marcha de un contenedor*


```
docker run -e=[variable de entorno=valor] [imagen a instanciar]
```

Con el objetivo de evitar la repetición del signo de igualdad, que hace la escritura un poco confusa, es posible utilizar el otro modo de gestión de las opciones de los comandos Docker, donde este símbolo simplemente se sustituye por un espacio.

#### *Añadir una variable de entorno (sintaxis equivalente).*

```
docker run -e [variable de entorno=valor] [imagen a instanciar]
```

A continuación se muestra un ejemplo con una variable, que serviría para especificar al contenedor la dirección IP de una máquina, necesaria para realizar su tarea, por ejemplo una llamada al servicio web alojado por esta máquina:

 Images/02RI131N.png

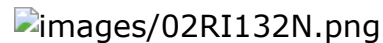
Para terminar, como normalmente resulta de utilidad pasar conjuntos similares de variables de entorno a varios contenedores, una opción muy práctica permite utilizar un archivo descriptivo como entrada.

#### *Añadir variables de entorno almacenadas en un archivo en la puesta en marcha de un contenedor*

```
docker run --env-file=[nombre del archivo que contiene los pares]  
[imagen a instanciar]
```

A continuación se muestra un ejemplo muy sencillo de utilización de esta opción:





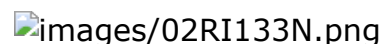
## h. Modificación de la hostname

Para terminar, una última opción muy simple y que permite modificar el nombre de la máquina (incluso si no se trata propiamente dicho de una máquina, sino de la designación del nombre de máquina en un contenedor de memoria estanco de una misma máquina).

### Modificar el hostname durante la puesta en marcha de un contenedor

```
docker run -h [hostname a utilizar] [imagen a instanciar]
```

El siguiente ejemplo muestra una ejecución de contenedor para el que los procesos hacen referencia al hostname tendrán `asterix` en lugar del identificador del contenedor, lo que es el comportamiento por defecto como se ve en el segundo comando. El tercero sirve solo para mostrar el hostname de la máquina host por referencia.



## 7. Manipulación de los contenedores

Después de algunas opciones sencillas, terminaremos este capítulo sobre los primeros pasos con Docker, con algunos comandos un poco más complejos, pero que nos permitirán perfeccionar su dominio de la herramienta antes de lanzarse a los siguientes capítulos, que tratan sobre usos más realistas - incluso industriales - de la herramienta.

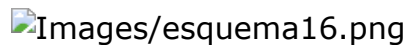
### a. Ejecución en modo bloqueante

Hasta ahora, en los ejemplos sobre todo hemos visto ejecuciones de contenedores Docker en modo bloqueante, es decir, que el shell a partir del que lanzamos el comando `docker run`, se ubica en modo de espera hasta el final del proceso y del contenedor Docker. En ese momento devuelve el control una vez se han detenido.

Este modo de funcionamiento está particularmente adaptado a las imágenes que proporcionan un shell como punto de entrada

por defecto (o cualquier proceso con una duración de vida limitada a su uso inmediato). En efecto, como el usuario va a entrar comandos en el contenedor, no es necesario que pueda seguir trabajando al mismo tiempo en la línea de comandos del host Docker. El hecho de impedir que continúen las operaciones en el shell host, puede permitir que se eviten errores en la operación, debido a la confusión entre los dos shells.

De manera esquemática, el circuito de control por las consolas entrantes y salientes, es el siguiente:



A continuación se muestra la descomposición de lo que esconde este sencillo comando `docker run -i -t ubuntu`:

- El shell host crea un contenedor y lo lanza.
- Las opciones `-i` y `-t` conectan las entradas y salidas del contenedor al shell de la máquina host.
- El contenedor arranca el proceso `/bin/bash` que utiliza estas entradas/salidas.
- El usuario de la shell host ha dado el control al shell cliente.
- Después de sus operaciones, en un momento dado entre el comando `exit`.
- El proceso se detiene, Docker lo detecta y detiene el contenedor.
- Si se activa la opción `--rm`, el cliente Docker elimina el contenedor justo después de haberlo detenido.
- El usuario recupera el control en el shell de la máquina host.


Si seguimos estas etapas en el esquema, solo se desarrolla una única línea y las líneas discontinuas que simbolizan una espera, terminan cuando todas las etapas se han ejecutado. Terminamos como hemos empezado: con un único proceso que es el de la shell host.

## **b. Ejecución en segundo plano**


Este modo de funcionamiento, ideal para los contenedores Docker que incorporan un proceso sobre el que el usuario encargado de la puesta en marcha va a interactuar, no se adapta del todo bien cuando el contenedor Docker arranca un proceso que usan otras personas, normalmente un servidor web o una base de datos.

Estos procesos no escuchan las entradas en la consola, sino mensajes en un puerto de red y estos últimos, pueden ser múltiples y ser completamente asíncronos respecto al ciclo de vida del contenedor.


Imaginemos que lanzamos un servidor web Nginx con el mismo comando `docker run -i -t nginx`, simplemente añadiendo un argumento que es necesario para que el puerto 80 expuesto por la imagen `nginx`, sea expuesto en la máquina host en el puerto 88 (el 80 normalmente ya está utilizado por el servidor web), a saber `-p 88:80`.

 Volveremos más en detalle sobre la gestión de los puertos en la siguiente sección.


¿Qué va a suceder? Como siempre dejamos el contenedor en modo bloqueante, el proceso va a funcionar bien, pero no devolverá el control:

 `images/02RI134N.png`


Vemos que la línea de comandos no ha reaparecido. Por el contrario, el servidor Nginx funciona bien:

 `images/02RI135N.png`

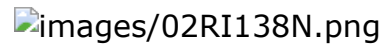
Si volvemos a la línea de comandos, vemos aparecer los logs correspondientes a las llamadas realizadas, en este caso un `GET http` en la página raíz (`/`):

 `images/02RI136N.png`

Pero si queremos retomar el control sobre el shell host y pulsamos `[Ctrl] C`, el proceso se va interrumpir y el contenedor se detiene, como podemos constatar ejecutando el comando `ps` (seguido del contenedor actual):

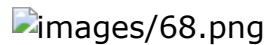
 `images/02RI137N.png`

También podemos lanzar el comando `ps -a` y verificar que el contenedor existe, pero que está en estado `Exited`:



Este estado se corresponde con el hecho de que el comando principal lanzado en el contenedor, ha terminado su tarea y ha salido con un código de resultado, el entero 0 que representa un resultado normal.

De repente, el servidor Nginx ya no responde más:



La representación gráfica del flujo de la línea de comandos que se corresponde con esta ejecución, es la siguiente:



Representamos todas las etapas, pero lo más importante es que, cuando el contenedor ha arrancado, ha activado al proceso `nginx` y este ha devuelto el control (líneas discontinuas). En efecto se trata como cualquier otro servidor web, escuchando el puerto que se le ha asignado y lanzando uno o varios procesos para tratar estas peticiones, pero no hay necesidad alguna de conservar el control sobre la shell a partir del que se ha ejecutado.

De golpe, el contenedor considera que ha terminado su trabajo de ejecución de proceso (después de todo, el servidor Nginx ha sido correctamente arrancado, en conformidad con lo que le hemos pedido) y por lo tanto, se detiene. De nuevo una vez más, si la opción `--rm` está presente, el contenedor también se destruye. Pero el simple hecho de detener el contenedor tiene un evidente efecto sobre el proceso `nginx`, que ve cómo se pierde su soporte y él también se detiene.

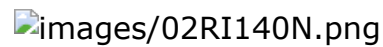
La respuesta a este problema es lanzar el contenedor en modo desatendido. Hablamos de esta manera de modo "segundo plano".

### Lanzar un contenedor en modo desatendido

```
docker run -d [imagen a instanciar]
```

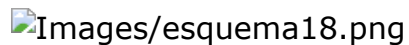
Gracias a este comando, el shell host retoma el control desde el final de la actividad de ejecución del contenedor, el servidor web

funciona como estaba previsto y una llamada del comando `ps` muestra que el contenedor está en estado activo (`up`):




La salida del comando `ps` muestra también que el puerto 80 ha sido correctamente redirigido sobre el 88.

Todo esto, se puede resumir de manera esquemática como sigue:



Esta vez, cuando se lanza el proceso `nginx`, Docker sabe por la opción `-d` que no debe pasar a la parada del contenedor, sino simplemente devolver el control a la línea de comandos host y dejar al contenedor vivir su vida. A partir de ese momento, habrá dos procesos que continúan: la shell host, así como el correspondiente al servidor Nginx en el contenedor.

 Si recibe un mensaje que contiene el texto `port is already allocated`, es posible que intente lanzar un segundo contenedor sobre el mismo puerto que un contenedor que no se ha detenido. Por lo tanto, hay conflicto y Docker se lo indica. En función del contexto, será necesario detener uno de los contenedores o elegir puertos diferentes.

### **c. Gestión correcta del ciclo de vida de los contenedores**


A continuación mostramos con un contenedor `nginx` correctamente lanzado, y que puede responder a peticiones en el puerto HTTP redirigido, y esto sin restricción relacionada con el ciclo de vida de la shell host. Precisamente esto es lo que buscamos conseguir, pero será necesario detener este contenedor un día u otro, solo por razones de mantenimiento o actualización de los archivos y operaciones expuestas.

En resumen, después de haber encontrado el medio para que el contenedor no se detenga después de la ejecución del proceso, ahora es necesario mostrar cómo detenerlo de manera explícita.


#### *Detener un contenedor*

```
docker stop [contenedor]
```

En nuestro caso, no habíamos llamado explícitamente al contenedor, por lo que podemos utilizar su nombre generado o su identificador, incluso de manera parcial:

images/02RI141N.png

De manera esquemática, lo que sucede es el siguiente encadenamiento:

Images/esquema19.png


El comando `stop` envía una señal `SIGTERM` al proceso, para evitar que deba detenerse. Esto último dispone de un periodo de gracia, durante el que puede consumir algunos últimos recursos para dejar su actividad limpiamente. Si, al cabo de este periodo el proceso sigue activo, se envía una segunda señal de tipo `SIGKILL`. Esta señal es mucho más brusca, en el sentido de que cortará bruscamente el proceso.

También es posible no dejar al proceso el tiempo necesario para detenerse correctamente, utilizando directamente el método `kill`.

### *Enviar una señal a un contenedor*

```
docker kill -s [código de la señal] [contenedor]
```

Por defecto, el código de la señal es `KILL`, lo que se corresponde con una parada del proceso destino.


 Preste atención con utilizar este método de parada de los contenedores de manera cautelosa. No plantea ningún problema para los contenedores de pruebas, pero no se debe utilizar en servidores de producción, donde las peticiones que se están ejecutando, no se deben destruir bruscamente.

Una de las etapas lógicas después de parar un contenedor, es su eliminación. Ya hemos mostrado varias veces el uso del comando `rm` para este fin, pero hay una opción interesante que merece una explicación.

### *Eliminar un contenedor de manera forzada*

```
docker rm -f [contenedor]
```

Añadir esta opción permite forzar la eliminación de un contenedor, enviando inicialmente la señal `KILL` al proceso incorporado. De esta manera, permite no tener mensajes de error sobre una eliminación, porque el contenedor todavía está activo. De hecho, el comando `rm -f` es equivalente al comando `kill` seguido de un comando `rm` simple.

 Observe que las opciones `--rm` y `-d` son incompatibles, lo que tiene cierta lógica, en el sentido de que la segunda consiste en abandonar la gestión del ciclo de vida del contenedor. Dando por hecho que será otra persona la que decida detener el contenedor, es lógico que este sea él el que se encargue, si es necesario, de eliminarlo y que el proceso de ejecución no tenga que intervenir más.

Para terminar con los comandos principales para la gestión del ciclo de vida, nos queda presentar los comandos de puesta en marcha de un contenedor detenido y de volver a arrancarlo.

#### *Poner en marcha un contenedor detenido*

```
docker start [contenedor]
```

#### *Volver a arrancar un contenedor en curso*

```
docker restart [contenedor]
```

El comando de reinicio es un atajo que permite lanzar `stop` y después `start`. Como para `stop`, es posible configurar el periodo de gracia con ayuda de la opción `-t`. De esta manera el siguiente comando intentará detener limpiamente el proceso servidor y, si este no se detiene al cabo de los 20 segundos, terminará con un `SIGKILL`, para después volver a lanzarlo.

#### *Reciclar un contenedor especificando el periodo de gracia*

```
docker restart -t 20 [contenedor]
```

La utilización típica de este comando es volver a lanzar un proceso reduciendo la memoria bloqueada o que se encuentra en cualquier condición detectada como anormal. El vocabulario utilizado habitualmente habla de "reciclaje de procesos".

Al contrario que las apariencias, se trata de un modo de funcionamiento totalmente normal: en efecto es muy complejo (y

por lo tanto muy costoso), garantizar que el código servidor no tiene fallos. Los criterios económicos dictan que es preferible hacer un reciclaje de vez en cuando y poner en batería varios contenedores que no se reciclarán al mismo tiempo, para garantizar una continuidad del servicio.

#### **d. Exposición de archivos**


Para volver a nuestro servidor Nginx, la sencilla visualización de una página por defecto es un uso extremadamente reducido. Por supuesto, podemos construir una imagen encima de la imagen nginx y añadir los archivos web en el lugar correcto, pero afortunadamente hay una funcionalidad que hace la exposición de estos archivos mucho más sencilla que todo esto. Se trata de la gestión de los volúmenes.

Más adelante volveremos en el detalle sobre los volúmenes, pero como primer enfoque vamos a utilizar la opción `-v` del comando `run`:

*Sustituir una sección de la arborescencia del contenedor por una ubicación host*

```
docker run -v [directorio host]:[directorio del contenedor] ...
```

En nuestro ejemplo, vamos a poner en correspondencia el directorio `/usr/share/nginx/html` (es decir, la localización por defecto de los archivos a exponer por Nginx) con `/home/swarm/www` (en la que hemos copiado los archivos descriptivos de la raíz del sitio web del autor):


images/02RI142N.png

El resultado se puede comprobar en un navegador:

images/69.png

Los archivos web depositados en `/home/swarm/www`, y en particular la página `index.html`, los utiliza el contenedor llamado `web` y que está basado en la imagen `nginx`. Por lo tanto, hemos enriquecido el funcionamiento por defecto de esta imagen, sin tener que crear otra imagen. Se trata de una de las funcionalidades más útiles de Docker.



 Usamos el valor adicional `:ro` del argumento `-v`, para especificar que se autoriza el acceso por el contenedor al directorio de la máquina host, pero únicamente en modo solo lectura. Se trata de una buena práctica para securizar el contenido de los archivos respecto a eventuales efectos colaterales del proceso en el contenedor.

## e. Supervisión de los contenedores


Nuestra última etapa en estos "primeros pasos con Docker", consiste en garantizar un primer nivel sencillo de supervisión de los contenedores. Nada masivo por el momento, sino solo dos comandos que permiten entender un poco mejor lo que pasa en un contenedor dado.

El primero está relacionado con los logs. Cuando arranca, Docker se conecta a los logs de un contenedor y permite realizar una redirección de estos para centralizarlos, dirigirlos a un lugar particular, etc. Los usos son múltiples pero en un primer momento, una gran parte de las veces la necesidad se centra en una visualización por línea de comandos.

### Visualizar los logs de un contenedor

```
docker logs [contenedor]
```

Tomemos el ejemplo de un servidor Nginx. Una vez puesto en marcha, llamaremos a una página inexistente:

 images/02RI144N.png

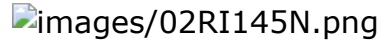
Los logs nos permiten ver en detalle lo que sucede. El navegador web se limita a mostrar un error 404. Por supuesto, el detalle depende del proceso que envía esta información. Por ejemplo, por defecto Nginx no devolverá el log de una llamada a la página raíz.

El segundo comando principal para la supervisión, permite seguir el proceso. La variedad puede sorprender, en el sentido que se ha explicado varias veces de que un contenedor contiene un proceso y solo uno. La realidad es que es importante tener un proceso y solo uno como punto de entrada, pero es imposible evitar que arranque otros procesos una vez que se encuentra en el contenedor.

## Explorar el proceso de un contenedor

```
docker top [contenedor]
```

Es lo que pasa con Nginx, que se ha iniciado en forma de proceso maestro durante la ejecución del contenedor, pero que inmediatamente ha creado un proceso de trabajo para descomponer sus responsabilidades de servidor web, como se muestra en la siguiente captura:

images/02RI145N.png

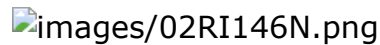
Hay otro comando que ha aparecido más recientemente, que permite seguir en tiempo real el consumo de recursos, como lo hace la herramienta `top` en Linux. Se trata del comando `stats`.

## **Seguir la evolución de los recursos utilizados por un contenedor**

---

```
docker stats [contenedor]
```

En nuestro ejemplo, el comando `docker stats web` genera un resultado similar al siguiente:

images/02RI146N.png

La visualización se refresca automáticamente. Es necesaria la combinación [Ctrl] C para retomar el control en la consola.

También podríamos citar el comando `events` entre las funcionalidades de supervisión, pero se reserva para un uso más experto. Para conocer más sobre el tema que desee, se recomienda al lector visitar la documentación de Docker, seguir los eventos correspondientes a las diferentes etapas del ciclo de vida de los contenedores y de las imágenes: <https://docs.docker.com/engine/reference/commandline/events/>

## Regreso a los primeros pasos

Este capítulo ha mostrado los principales comandos para controlar Docker y abordar, de manera serena, sus usos más industriales.

Sin embargo, es conveniente observar que, aunque por el momento estemos únicamente en fase de iniciación, los usos realizados ya son importantes:

- Hemos creado contenedores sellados con un OS potencialmente diferente al de la máquina host.
- Hemos lanzado procesos potencialmente peligrosos en un entorno de pruebas, haciendo de esta manera que sean inofensivos.
- Hemos expuesto archivos estáticos a un servidor web y esto, sin tener que gestionar una instalación compleja.
- Hemos iniciado y detenido más de una docena de entornos, todo en apenas pocos minutos.

Una de las grandes fortalezas de Docker es permitir, desde las primeras operaciones, realizar operaciones de alto valor pero potencialmente complejas de realizar sin esta herramienta. Además, podemos establecer una comparación con los enfoques de tipo cloud.

Debido a su enfoque destinado al alquiler, Amazon, Azure y otros clouds permiten que proyectos de pequeño tamaño accedan a los recursos informáticos, que estarían fuera de su alcance financieramente hablando.

De la misma manera, Docker permite montar arquitecturas de software complejas de manera sencilla, lo que abre la vía a determinados proyectos que no podrían disponer de las competencias de instalación y configuración de Nginx, MongoDB, etc. Las imágenes disponibles en Docker Hub permiten a todos disponer gratuitamente de la mejor experiencia de implantación de las aplicaciones recientes y concentrarse en su propio valor añadido (contenido en el caso de un servidor web, procesos de negocio para una base de datos, etc.).

El atractivo del enfoque Docker, es que simplifica de tal manera el uso de la tecnología que, en la mayor parte de los casos, la pregunta sobre los requisitos previos o de infraestructura no supone ningún problema y ya no se plantea. Por ejemplo, hemos

explotado un servidor Nginx sin saber en qué distribución se basaba su imagen. Todo lo que nos interesaba era exponer los archivos web en un puerto HTTP: Docker nos ha permitido ocuparnos solo de esto y nada más.

Esta separación entre la exposición de un servicio y su implantación, es uno de los fundamentos de las arquitecturas orientadas a servicios. En los siguientes capítulos, utilizaremos justamente una arquitectura SOA para mostrar de manera práctica los usos más avanzados de Docker.

## 3. CREACIÓN DE SUS PROPIAS IMÁGENES

### Creación manual de una nueva imagen

En el capítulo anterior, hemos estudiado el funcionamiento básico de Docker, utilizando imágenes preparadas con antelación para nuestros ejemplos: la imagen `hello-world` para las primeras pruebas, la imagen `ubuntu` para los enfoques interactivos por la línea de comandos y para terminar, la imagen `nginx` para las pruebas que implican un proceso servidor. Estas imágenes se han recuperado en línea del registro Docker Hub.

Aunque esta manera de funcionar pueda ser suficiente en las implantaciones extremadamente sencillas, llega un momento donde se hace necesario crear sus propias imágenes para evolucionar hacia niveles mayores de complejidad. Este es el tipo de operaciones que vamos a ilustrar en el presente capítulo.

#### 1. Instalación de un software en un contenedor

El enfoque más sencillo para crear su propia imagen es utilizar el comando `commit`, que se ha dejado entrever en el capítulo anterior, para persistir el estado de un contenedor en forma de una nueva imagen, después de haberle añadido las modificaciones necesarias.

El siguiente ejemplo consiste en utilizar una imagen `ubuntu` como base, instalar MongoDB (una base de datos no relacional) en el contenedor y después, persistirla como una imagen reutilizable.


Lance en modo interactivo un contenedor basado en la imagen `ubuntu`.

En la línea de comandos escriba los siguientes comandos, necesarios para la instalación de MongoDB. El código proviene de la documentación (<http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/>) y se copia a continuación:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv 7F0CEB10
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release
```

```
-sc)"/mongodb-org/3.0 multiverse" | sudo tee  
/etc/apt/sources.list.d/mongodb-org-3.0.list  
sudo apt-get update  
sudo apt-get install -y mongodb-org
```

Una vez realizada la instalación, lance el servicio MongoDB, arrancando manualmente el demonio. Para ello use el comando `mongod --config /etc/mongod.conf &`:

 images/03RI01.png

La llamada al comando `ps` permite verificar que el proceso `mongod` se ha lanzado correctamente. El comando anterior se lanzará con el símbolo `&` al final del comando. Esto permite devolver el control y hacer que el proceso MongoDB se ejecute como tarea desatendida. La opción `--config` permite apuntar al archivo de configuración de MongoDB, que se deberá modificar si es necesario.

Una vez que se lance el proceso, podemos utilizar el cliente `mongo`, también instalado por el procedimiento utilizado. Podemos conectarnos al servidor, insertar un dato y lanzar una consulta, todo ello para validar que todo funciona correctamente:

 images/12.png

Elimine la colección `personas` para que la imagen que vamos a crear no se "contamine", utilizando el comando `db.personas.drop()`.

Salga del cliente MongoDB usando el comando `exit`.

Salga del contenedor Docker con el comando `exit`, que detendrá la shell alojado.

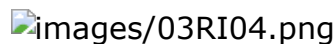
El cambio de la línea de comandos debe mostrar que hemos vuelto a la máquina host, como se puede ver en la siguiente captura:

 images/13.png

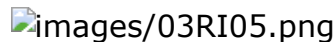
## 2. Persistencia de la imagen para un uso futuro

Ahora vamos a pasar a la segunda etapa, que consiste en crear una imagen a partir del contenedor. A este no se le había asignado

un nombre, por lo que debemos encontrar cómo identificarlo empleando el comando `ps` de Docker:

images/03RI04.png

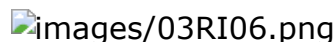
A continuación se utiliza el comando `commit` para crear una imagen y después, validamos su creación llamando al comando `images`:

images/03RI05.png

### 3. Utilización de la imagen creada

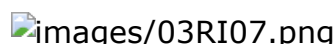
Se va a utilizar la imagen llamada `jpgouigoux/mongodb`, para lanzar un nuevo contenedor (el anterior, que nos ha servido para crear la imagen, se puede eliminar sin problemas). Con el objetivo de facilitar las operaciones, vamos a llamar a este contenedor `mongo` en todos los ejemplos que siguen.

Empezamos arrancando el contenedor en modo interactivo, de manera que se verifique que el servidor MongoDB está presente:

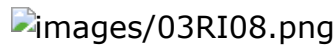
images/03RI06.png

En parte es el caso (vemos correctamente el ejecutable `mongod` en `/usr/bin`), pero el proceso no se ha iniciado como lo hizo cuando hemos quitado el anterior contenedor. El comportamiento es el esperado, es decir, que todas las modificaciones en el disco se han realizado correctamente en la imagen, pero respecto al proceso, un nuevo contenedor lanza un nuevo proceso y el comando por defecto de nuestra nueva imagen siempre es `/bin/bash`. Cuando el contenedor arranca, tomamos el control en modo interactivo pero para que el servidor MongoDB sea accesible, es necesario lanzar de nuevo el comando `mongod --config /etc/mongod.conf`.

Este modo de funcionamiento no es recomendable: un mecanismo correcto de despliegue no incluye etapas manuales. Una primera manera de lanzar automáticamente el proceso es añadir el comando en el archivo `/etc/bash.bashrc`:

images/03RI07.png

Para comprobar la diferencia, es necesario relanzar una shell, en este caso en un contenedor. Atención, no olvide validar los cambios en la imagen, porque corre el riesgo de no ver ninguna modificación.



De esta manera, cuando lanzamos el contenedor `mongo` (no olvide eliminar el contenedor anterior), el proceso se ha lanzado correctamente. Puede comprobarlo con el comando `ps`.

## 4. Conexión desde la máquina host

Para terminar con una conexión un poco más realista, conviene conectarse a la base MongoDB, no desde el contenedor en sí mismo, sino desde el exterior. Por ejemplo, desde la máquina host.

La primera etapa para esto es hacer que el servidor MongoDB sea accesible desde el exterior (como medida de seguridad, solo escuche por defecto las llamadas locales):

Si ha salido del contenedor `mongo`, elimínelo y vuelva a lanzar uno con el mismo nombre.

Edite el archivo `/etc/mongod.conf` (con la herramienta `vi`, por ejemplo).

Localice la línea `bind_ip=127.0.0.1` y empiece añadiendo un símbolo `#` delante (comando `i` para insertar en `vi`).

Guarde el archivo (tecla `escape` y después `:wq` y tecla `intro` en `vi`).

Escriba `exit`.

Una vez que se detiene el contenedor, lance el mismo comando `commit` que antes.

Elimine el contenedor con el comando `docker rm mongo`.

Vuelva a lanzar un nuevo contenedor en modo interactivo con el comando `docker run -it jpgouigoux/mongodb`.

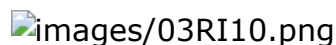
La segunda etapa es instalar el software cliente de MongoDB en la máquina desde la que queremos conectarnos, por ejemplo la máquina host de Docker. Si la máquina utiliza Ubuntu, lo más



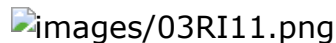
sencillo es utilizar el administrador del Centro de software para esto:



La tercera etapa antes de la conexión, consiste en encontrar la dirección IP expuesta por el contenedor Docker, de manera que apunte al servidor correcto desde el cliente. Podríamos lanzar el comando `ifconfig eth0` en el contenedor, pero parece más lógico utilizar el comando `inspect` de Docker para encontrar esta información desde la máquina host:



Ahora es posible la conexión desde la máquina host, como sigue:



## 5. Seguimiento de las operaciones

El modo de funcionamiento implementado, todavía está lejos ser satisfactorio. La captura anterior muestra claramente los avisos sobre el hecho de que el proceso de servidor se lanza sobre el usuario `root`, lo que es una muy mala práctica desde el punto de vista de la seguridad.

Además, nuestro contenedor se lanza en modo interactivo, lo que no es muy lógico, visto el tipo de uso con un servidor de base de datos en nuestro ejemplo. Y si cerramos la Shell, se detiene el contenedor y también lo hace MongoDB, así que terminamos manteniendo una ventana abierta solo para poder acceder a un servicio. No es muy limpio...

Por supuesto, podríamos ajustar estos problemas creando nosotros mismos un usuario dedicado para el servidor MongoDB, lanzando de otra manera el proceso servidor, etc. Pero esto supondría bucear mucho en la documentación de instalación de MongoDB, sin tener garantías de realizar una instalación tan limpia como la implementada por un especialista de esta base de datos NoSQL.

Esta implantación limpia ya existe, en forma de imagen Docker oficial para MongoDB. Además, las personas encargadas de esta imagen ofrecen la "receta" en forma de un archivo que contiene

todos los comandos necesarios para una instalación perfecta. ¿Por qué hacerlo de otra manera?

# Utilización de un Dockerfile

## 1. Interés de los archivos Dockerfile

La "receta" de la que acabamos de hablar, se presenta en forma de archivo de texto llamado `Dockerfile`, que utiliza una gramática particular de Docker. Este archivo contiene todas las operaciones necesarias para la preparación de una imagen Docker. De esta manera, en lugar de construir una imagen por medio de operaciones manuales en un contenedor, seguida de un comando `commit` (incluso varios si procede por etapas), como hemos hecho anteriormente, vamos a poder compilar una imagen desde una descripción textual de estas operaciones.

A continuación se muestra, por ejemplo, el contenido del archivo `Dockerfile` correspondiente a la imagen oficial MongoDB, tal y como se puede encontrar en el registro Docker Hub a la fecha de escritura de este libro (versión 3.0 de MongoDB, disponible en [https://registry.hub.docker.com/\\_/mongo/](https://registry.hub.docker.com/_/mongo/)):

```
FROM debian:wheezy

# add our user and group first to make sure their IDs get assigned
consistently, regardless of whatever dependencies get added
RUN groupadd -r mongodb && useradd -r -g mongodb mongodb

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        ca-certificates curl \
        numactl \
    && rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
RUN gpg --keyserver pool.sks-keyservers.net --recv-keys
B42F6819007F00F88E364FD4036A9C25BF357DD4
RUN curl -o /usr/local/bin/gosu -SL
"https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg
--print-architecture)" \
    && curl -o /usr/local/bin/gosu.asc -SL
"https://github.com/tianon/gosu/releases/download/1.2/gosu-$(dpkg
--print-architecture).asc" \
    && gpg --verify /usr/local/bin/gosu.asc \
    && rm /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu

# gpg: key 7F0CEB10: public key "Richard Kreuter
<richard@10gen.com>" imported
RUN apt-key adv --keyserver pool.sks-keyservers.net --recv-keys
492EAFE8CD016A07919F1D2B9ECBEC467F0CEB10
```

```
ENV MONGO_MAJOR 3.0
ENV MONGO_VERSION 3.0.1

RUN echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-org/MONGO_MAJOR main" > /etc/apt/sources.list.d/mongodb-org.list

RUN set -x \
    && apt-get update \
    && apt-get install -y mongodb-org=$MONGO_VERSION \
    && rm -rf /var/lib/apt/lists/* \
    && rm -rf /var/lib/mongodb \
    && mv /etc/mongod.conf /etc/mongod.conf.orig

RUN mkdir -p /data/db && chown -R mongodb:mongodb /data/db
VOLUME /data/db


COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

Las líneas se corresponden con las actividades manuales que hemos realizado anteriormente, resaltadas en negrita, para mostrar la riqueza adicional del archivo descriptivo de la imagen oficial. En particular, encontramos:

- La implementación de un usuario y de un grupo dedicados.
- La asignación de los permisos correspondientes a los directorios, para los que esta operación es necesaria.
- La utilización de un `ENTRYPOINT` dedicado, que permite especificar el script que se debe lanzar durante la puesta en marcha del contenedor (volveremos de manera más extensa sobre este comando más adelante).
- La implantación de un proceso por defecto, diferente de la shell.

Una vez más, habría sido posible encontrar todas estas operaciones buscando en la documentación o en los foros. Pero la presencia de un `Dockerfile` generado por los expertos de MongoDB al mismo tiempo que dejándonos validar los comandos, es una enorme ventaja de Docker.

 Sigue siendo necesaria una mirada crítica sobre los archivos `Dockerfile`, porque se corre el riesgo de no volver a estar seguro salvo en el caso de la sencilla instanciación de una imagen de procedencia desconocida. En particular, cualquier llamada a las URL externas (normalmente por `curl`), debería ser cuidadosamente comprobada.

## 2. Utilización de un archivo Dockerfile

En el caso de una imagen oficial, aunque el archivo `Dockerfile` sobre todo sirve a la persona que lo crea realizar la imagen y mostrar públicamente la manera en la que se realiza (la unión entre los dos queda garantizada en particular por el hecho de que es Docker Hub el que se ocupa de compilar las imágenes), es posible utilizar este archivo para crear una imagen.

En la máquina host, cree un directorio llamado por ejemplo `mongogb`.

Sitúese en este directorio.

Vuelva a copiar los dos archivos que recuperó de:

<https://github.com/docker-library/mongo/tree/master/4.1>

Asigne permisos de lectura a los dos archivos y los de ejecución a `docker-entrypoint.sh`.

El segundo archivo (además de `Dockerfile`), es el archivo `docker-entrypoint.sh` al que hace referencia `Dockerfile` (comando `ENTRYPOINT`, que vamos a explicar más adelante). Es necesario para que el contenedor lance el servidor MongoDB en el usuario dedicado, en lugar de `root`. Para esto utilice la herramienta `gosu`, que se ha instalado anteriormente usando un comando `RUN` del `Dockerfile`.

La parte del contenido de este archivo que se corresponde con este comando, es la siguiente:

```
#!/bin/bash
set -Eeuo pipefail

(...)

exec gosu mongod " $BASH_SOURCE " "$@"

(...)

exec "$@"
```

Lance el comando de compilación de la imagen.

### Compilar un archivo Dockerfile

```
docker build [ubicación de Dockerfile]
```




El proceso puede llevar algunos minutos, en función del ancho de banda disponible para el acceso a Internet, así como de la presencia inicial de determinadas capas de la imagen. Al final del proceso, debería aparecer un mensaje `Successfully built` seguido de un identificador, lo que significa que el comando ha tenido éxito.

Por el momento, la imagen creada no tiene un nombre muy explícito. La renombramos con el comando `docker tag`:



### 3. Resultados de la utilización de un Dockerfile completo

 A partir de ahora, utilizamos un contenedor que llama a un volumen. Vamos a explicar más adelante esta noción, pero es importante que a partir de ahora cualquier eliminación del contenedor se realice con la opción `-v`.

La imagen creada por este método - y que es totalmente idéntica a la imagen oficial que podríamos haber descargado directamente por el registro Docker Hub - es de mejor calidad. La ejecución en segundo plano es sencilla y no necesita intervención compleja en los archivos de la Bash.

Una llamada al comando `docker ps` muestra que el contenedor está correctamente activo, con el proceso servidor a la escucha:



Además, la conexión ya no muestra las advertencias sobre el hecho de que el servidor funcione en el usuario `root` y por tanto, con plenos poderes sobre el contenedor (aunque la otra advertencia siempre esté presente):



Un análisis más completo del `Dockerfile` muestra todavía algunas sorpresas, que vamos a descubrir en la siguiente sección, en particular la gestión de la persistencia de MongoDB por un mecanismo llamado "volumen".


## 4. Anatomía de un archivo Dockerfile

En el capítulo anterior, ya se había procedido un rápido análisis de un archivo `Dockerfile`, en este caso de la imagen `hello-world`. Este estaba en la cúspide de la simplicidad de la imagen, únicamente con tres líneas. Aquí vamos a analizar un poco más en detalle el `Dockerfile` de la imagen `mongo`, para entrar un poco más profundamente en la gramática y las funcionalidades del lenguaje de descripción de una imagen Docker.

### a. FROM

```
FROM debian:wheezy-slim
```

Los archivos `Dockerfile` empiezan siempre con el comando `FROM`, que permite definir una imagen básica encima de la que se construirá la nueva imagen. En nuestro ejemplo, la imagen básica es una Debian, versión Wheezy, en su versión de tamaño reducido (con el sufijo `slim`, que quiere decir "delgado" en inglés).

 Las imágenes "slim" son imágenes básicas Docker con un sistema voluntariamente aligerado de módulos que normalmente, no son necesarios en los usos de contenedores. Esto permite aligerar las descargas. Algunas distribuciones incluso se crean específicamente por estas necesidades de ligereza, como por ejemplo las imágenes Alpine.

Si esta imagen no está presente en la máquina host, se descarga durante la ejecución del comando `build`. Cuando hemos listado las imágenes para renombrar la nueva creada un poco más atrás, vimos la imagen `debian` con el tag `wheezy`.

En los raros casos en que una imagen está construida sin partir de una imagen existente, la gramática a utilizar es `FROM scratch`. El nombre no se corresponde realmente con una imagen básica, sino con la expresión *from Scratch*, que en inglés significa "desde cero", por lo que se adapta particularmente bien en este caso.

## b. RUN

El comando `RUN` es uno de los más utilizados en el `Dockerfile`. Permite lanzar cualquier proceso que va a participar en la elaboración de la imagen que se está compilando.

En nuestro ejemplo, el comando `RUN` se utiliza varias veces para:

- Lanzar un comando `curl`, que permite recuperar archivos en Internet.
- Utilizar `echo` para escribir datos en un archivo que se encuentran en la imagen.
- Manipular los diferentes comandos del administrador de paquetes `Aptitude`, para instalar determinados módulos.
- Gestionar los archivos y directorios en función de las necesidades.

El archivo `Dockerfile` se corresponde con la imagen oficial `mongo`, solo contiene uno de los dos tipos de llamadas del comando `RUN`, a saber, las que utilizan un único argumento. En este caso, la cadena se corresponde con el comando que se pasará al shell `/bin/sh` y este se lanzará con la opción `-c`, para aceptar esta entrada por la línea de comandos en lugar de la consola de entrada.

Existe una segunda sintaxis de utilización de `RUN`, que utilizan varios argumentos.

### Lanzar una operación RUN en formato exec


```
RUN ["ejecutable", "argumento 1", ..., "argumento N"]
```

Esta manera de proceder se debe utilizar si desea utilizar otra shell, escribiendo por ejemplo:

```
RUN ["/bin/bash", "-c", "echo mensaje"]
```

También sirve en el caso en el que simplemente es inútil pasar por una shell (normalmente porque no es necesaria la sustitución de variable). Entonces, la gramática será simplemente:

```
RUN ["echo", "mensaje"]
```

 Más adelante volveremos sobre la diferencia entre el enfoque shell y el enfoque exec, que se encuentra en los operadores `CMD`



y ENTRYPOINT.

### c. ENV

```
ENV MONGO_MAJOR 3.0
ENV MONGO_VERSION 3.0.15
```

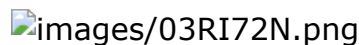
El comando `ENV` permite determinar las variables de entorno, que estarán disponibles tanto para el resto de la operación de compilación de `Dockerfile`, como para contenedores que se lanzarán desde la imagen generada.

Para explicar este importante concepto, a continuación se muestra en primer lugar un ejemplo de utilización dentro de `Dockerfile`:


```
RUN set -x \
&& apt-get update \
&& apt-get install -y \
  ${MONGO_PACKAGE}=${MONGO_VERSION} \
  ${MONGO_PACKAGE}-server=${MONGO_VERSION} \
  ${MONGO_PACKAGE}-shell=${MONGO_VERSION} \
  ${MONGO_PACKAGE}-mongos=${MONGO_VERSION} \
  ${MONGO_PACKAGE}-tools=${MONGO_VERSION} \
&& rm -rf /var/lib/apt/lists/* \
&& rm -rf /var/lib/mongodb \
&& mv /etc/mongod.conf /etc/mongod.conf.orig
```

Esta utilización permite centralizar los argumentos y de esta manera, hacer más fácilmente escalable los `Dockerfile`, evitando en este caso concreto sustituir una versión y olvidar otra ocurrencia posterior en el archivo.

A continuación se muestra la manifestación del segundo impacto del comando `ENV`:



Si arrancamos un contenedor, encontramos las variables de entorno en su contexto de ejecución.

 Por lo tanto, es posible pasar variables de entorno únicamente durante la ejecución del contenedor, sin que formen parte de la definición de la imagen. La opción `--env` del comando `docker run` está disponible para este efecto.

## d. VOLUME

```
RUN mkdir -p /data/db /data/configdb && chown -R mongodb:mongodb  
/data/db /data/configdb  
VOLUME /data/db /data/configdb
```

Se dedicará una sección completa en otro capítulo a la gestión de los volúmenes, que es un aspecto importante. En esta sección, simplemente vamos a explicar la problemática de los datos, cómo podemos aprovechar la definición de volumen anterior para no perder datos de MongoDB y después, volver brevemente sobre la importancia de la opción `-v` del comando de eliminación de los contenedores.

En la sección Exposición de archivos del capítulo anterior, hemos abordado rápidamente la noción de volumen, asociando a una imagen `nginx` un directorio `www` que contiene los archivos estáticos a exponer por el servidor web. Esta operación permite conservar una imagen neutra y no tener que reconstruir o validar las modificaciones después de cada cambio de los archivos web.

En caso de un servidor base de datos como MongoDB, todavía es más importante que los archivos de datos estén desacoplados del contenedor de base de datos en sí mismo. En efecto, aunque no sea problemático relanzar el proceso de servidor `mongod`, partiendo de cero para los datos contenidos (y perder todos los datos escritos por los usuarios) es un enorme problema potencial.

Ilustramos este problema mostrando dos ejecuciones. La primera insertando datos y el segundo contenedor comprobando su ausencia:



Con el objetivo de producir un resultado sin perder datos, el cambio es mínimo: es suficiente con hacer corresponder el volumen `/data/db` expuesto por la imagen con un directorio, en el que los datos estén securizados. En nuestro caso, vamos a utilizar un directorio `datos`, situado en la máquina `host`.


La siguiente operación, como se muestra más abajo, solo incluye una opción `-v` en los comandos `run` (así como la creación del directorio), pero conduce a un resultado importante: el segundo contenedor ve los datos escritos por el primero:



De nuevo una vez más, aquí solo mostramos un primer enfoque de los volúmenes. Se dedica una sección completa más adelante. El objetivo era solo mostrar para qué sirve la palabra clave `VOLUME` en el archivo `Dockerfile`, en este caso especificar una parte del sistema de archivos del contenedor al que se le permite estar expuesto al exterior. De esta manera es posible hacer que se corresponda con un directorio de la máquina host como acabamos de hacer o incluso, enlazar con un contenedor dedicado a la gestión de un volumen, como veremos más tarde.

¿Qué sucede cuando olvidamos apuntar a un volumen en un directorio dado? Simplemente Docker lo hace por nosotros, creando un directorio controlado por su cuenta en la máquina host, durante la puesta en marcha del contenedor.

Una llamada a `docker inspect` permite encontrar la ubicación de este directorio y su contenido, gestionado por MongoDB:

 `images/03RI19.png`

Este directorio no se reutilizará para un segundo contenedor, porque Docker rompería la regla de estanqueidad. Pero, por defecto, tampoco fue eliminado, para dar una posibilidad de recuperar los datos si es necesario. Es una seguridad apreciable, pero es fácil encontrarse con muchos de volúmenes llamados huérfanos, es decir, del espacio en disco consumido por estos volúmenes creados automáticamente, pero no eliminados cuando el contenedor se detiene. Por esta razón se recomienda, cuando se experimenta con Docker, utilizar la opción `-v` en `docker rm`, de manera que los volúmenes se eliminen automáticamente cuando no se utilicen más por ningún contenedor.

En el archivo `Dockerfile` mostrado, se define un segundo volumen (`/data/configdb`) que permite apuntar MongoDB a un contenido de configuración usando el mismo mecanismo que para los datos.

## **e. COPY**

```
COPY docker-entrypoint.sh /usr/local/bin/
```

El comando `COPY` permite recuperar los archivos locales de la máquina en la que se lanza el comando `docker build` e integrarlos en la imagen que se está creando. En nuestro ejemplo, el archivo `docker-entrypoint.sh` estaba en el

mismo lugar que el archivo `Dockerfile`, entregado por los encargados de mantener la imagen oficial de MongoDB. Como este archivo es necesario para la ejecución cuando se lanza el contenedor, el comando `COPY` lo copia en un directorio y crea un enlace desde la raíz de la imagen compilada.

En la práctica, los archivos que están en la misma ubicación que el archivo `Dockerfile`, se copian en lo que se llama el contexto y por tanto, se cargan por el comando `build`. Por lo tanto, es importante no ubicar su archivo `Dockerfile` en un directorio con archivos que no serán útiles en la construcción de la imagen. Lo más sencillo es crear un directorio dedicado, como se puede observar en las imágenes del registro, que se construyen a partir de un almacén GitHub que solo contiene lo necesario.

Algunas veces es difícil obtener un directorio completamente libre de cualquier archivo inútil, justamente porque el mismo directorio es el que se utiliza para los desarrollos por Git. El truco consiste en añadir un archivo llamado `.dockerignore`, en el mismo lugar que el archivo `Dockerfile` y declarar en el primero los archivos y directorios que se deben ignorar durante la carga del contexto (los símbolos `*` y `?` se pueden utilizar con su significado usual de sustitución de un conjunto cualquiera de caracteres o cualquier carácter único).

## f. ENTRYPOINT

```
ENTRYPOINT ["/docker-entrypoint.sh"]
```

La palabra clave `ENTRYPOINT` permite definir el comando que se va a ejecutar durante la puesta en marcha de un contenedor, es decir, el proceso primario que este último portará. Como para el comando `RUN`, el funcionamiento es dual:

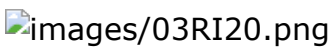
- En la gramática `ENTRYPOINT` seguido por una tabla JSON (como en nuestro ejemplo anterior, aunque la tabla se reduzca a una sola entrada), el primer argumento contiene el comando a ejecutar y los siguientes, los argumentos de este comando.
- En la gramática `ENTRYPOINT` seguida de una cadena, se arranca un shell y el contenido de esta cadena se le pasa para su interpretación. Atención, en este caso, las señales enviadas al contenedor por medio del comando `kill`, no se pasarán.

No hay interés particular en detallar todo el contenido del archivo `docker-entrypoint.sh`, que se ha copiado justo antes en la imagen (lo que es un requisito previo para que se pueda utilizar aquí).

Un punto interesante respecto a Docker, es que el archivo termina por la siguiente instrucción que hace que, si no se ha realizado ninguna condición, lo que se ejecuta es el comando que se haya pasado como argumento:


```
exec "$@"
```

Esto es lo que nos permite, por ejemplo, escribir el siguiente comando:



En el ejemplo anterior, hemos entrado en el script `docker-entrypoint.sh`, pero pasándole `/bin/bash` como argumento. Como esto no se correspondía con ninguna de las condiciones `if`, se lanzará la última línea, es decir, `exec "$@"`.


La segunda parte de la gramática expresa que se debe recuperar todos los argumentos (`$1` se corresponde con el primero, `$2` con el segundo, etc. y `$@` es la lista completa). Por lo tanto, lo que se ejecuta es `exec /bin/bash`, lo que explica que hemos podido tomar el control por la línea de comandos (con la opción `-it` en `docker run`, por supuesto), y guardar el contenido del archivo del contenedor lanzado. Se comprueba que después de `exit` se detiene el contenedor, lo que es lógico porque estábamos en modo interactivo y el proceso `/bin/bash` (que era el proceso principal), ha terminado.



Es una buena práctica que un contenedor solo albergue un único proceso (evidentemente, aparte de los procesos correspondientes a herramientas, cuya misión es diagnosticar el funcionamiento, por ejemplo). Este proceso tiene por número de PID 1 y es el que recibirá las señales enviadas por el comando `kill`. Esta observación explica en particular la limitación de la puesta en marcha por una shell, que hemos abordado anteriormente: como es la shell que recibe las señales, el ejecutable que ha arrancado por ejemplo no recibirá nada, durante un comando `docker stop`.

Si lanzamos un contenedor en la imagen MongoDB sin indicar argumentos, se utilizará el indicado por la palabra clave `CMD` del `Dockerfile`. Sin anticipar demasiado, en nuestro ejemplo su valor es `mongod`.

Es posible modificar el valor de `ENTRYPOINT` durante la ejecución del contenedor, utilizando la opción `--entrypoint`, pero esta no es una buena práctica. Se supone que `ENTRYPOINT` es la parte estable del comando de puesta en marcha del proceso a ejecutar. La parte variable, se corresponde con todo lo que se encuentra al final del comando `docker run` (justo después del nombre de la imagen a utilizar). En la mayoría de los casos, se utilizarán los argumentos por defecto especificados por la palabra clave `CMD` en el `Dockerfile`.

 Observe también que la lógica obliga a tener solo un `ENTRYPOINT` en un archivo `Dockerfile`. En la práctica, Docker no provoca ningún error si hay varios, pero solo tendrá en cuenta el último de ellos.

## g. EXPOSE

```
EXPOSE 27017
```

La palabra clave `EXPOSE` permite exponer un puerto de red al exterior, algo parecido a lo que hace la palabra clave `VOLUME`, con las secciones del sistema de archivos. De la misma manera que no es obligatorio conectar el volumen a un directorio de la máquina `host`, tampoco lo es poner en correspondencia el puerto expuesto con un puerto de la máquina `host`, pero este comando permite hacerlo, utilizando la opción `-p` del comando `docker run`.

Ya habíamos mostrado un primer ejemplo de utilización de esta opción durante la operación de la imagen `Nginx` en el capítulo anterior y volveremos más en detalle en un capítulo posterior, sobre la gestión de la red con Docker.

## h. CMD

```
CMD ["mongod"]
```

Ya hemos abordado el papel de la palabra clave `CMD`, durante la explicación de `ENTRYPOINT`: pasa al proceso especificado por `ENTRYPOINT` los argumentos por defecto, en ausencia de argumentos especificados explícitamente al final del comando `docker run`.

En nuestro ejemplo sobre la imagen oficial de MongoDB, el valor era `mongod`. Se selecciona la segunda condición `if` y la ejecución termina por:

```
exec gosu mongod "$@"
```

La lista de argumentos era `mongod`, por lo que el comando lanzado era el siguiente:

```
exec gosu mongod mongod
```

En la práctica, esto lanza el proceso `mongod` bajo la identidad del usuario `mongod`, en resumen, arranca nuestro servidor en un modo más seguro arrancándolo en `root`.

Con el objetivo de explicar bien la importante unión de `CMD` y `ENTRYPOINT`, volveremos con un ejemplo muy sencillo.

## 5. Nuestro primer Dockerfile

Este ejemplo será el de una imagen que, una vez instanciada en forma de contenedor, emitirá señales textuales a la consola de salida a intervalos regulares. El mensaje se configurará durante la puesta en marcha del contenedor, con un valor por defecto asignado por `CMD`. Utilizaremos `ENTRYPOINT` de tal manera que la emisión del mensaje se pueda detener y después volver a comentar.

### a. Creación y prueba del script

Antes incluso de hablar del archivo `Dockerfile`, vamos a crear un script shell que permite emitir un mensaje normal.

Cree un directorio llamado por ejemplo `repeater`, en el que se almacenen todos los archivos que vamos a utilizar.

Sitúese en este directorio.

Cree un archivo `heartbeat.sh` e inserte en él el siguiente

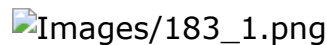
contenido:

```
#!/bin/bash
if [ -z "$HEARTBEATSTEP" ]; then
    echo "La variable de entorno HEARTBEATSTEP se debe evaluar"
    return 1
fi

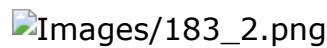
while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$HEARTBEATSTEP";
done
```

Guarde el archivo.

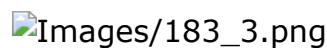
Una primera prueba nos permite validar el comportamiento esperado si no existe la variable `HEARTBEATSTEP`, que ajusta la duración entre cada mensaje:



Para una prueba funcional, podemos pasar temporalmente la variable de entorno, como se muestra a continuación:



Como información, también es posible fijar la variable de entorno de una sola vez en la sesión de la shell actual y esto es lo que se pasará cuando inyectemos el valor de la variable de entorno por el `Dockerfile`:




## b. Creación de Dockerfile

La siguiente etapa consiste en escribir un `Dockerfile` que nos va a permitir crear una imagen, lanzando el script con la configuración correcta. Lo que sigue en esta sección describe cada una de las líneas que componen este archivo, que situaremos junto al anterior, en el directorio `repeater`.

En primer lugar, usamos la última versión; el interés centrarse en una versión particular para evitar que los efectos sean casi nulos en nuestro ejemplo:

```
FROM ubuntu:latest
```



 La elección de una imagen básica es un aspecto complejo, que pueda depender de muchos factores y del uso que se vaya a hacer de su imagen. Para la mayor parte de los usos estándares, existen imágenes ligeras como Alpine, CoreOS y muchas otras. Es importante elegir la imagen básica más ligera posible entre las que le ofrecen las funcionalidades deseadas. Además, si crea varias imágenes (lo que seguramente sea el caso en las arquitecturas de micro-servicios), entonces hay una ventaja adicional de elegir una imagen básica idéntica para todos sus servicios, incluso si es un poco más densa, porque se cacheará una única vez para todas sus imágenes. En el caso del presente ejemplo, es necesario elegir una imagen más completa, porque las operaciones no se realizan de manera tradicional en un servidor. Por tanto, basar el contenido en una imagen Alpine provocaría un error de ejecución, ya que un archivo necesario no estaría presente.

Después tenemos información sobre la persona que mantiene esta imagen. Este comando es opcional, pero se recomienda si comparte la imagen. El formato de escritura es el siguiente:

```
LABEL maintainer="jp.gouigoux@free.es"
```


Algunos archivos Dockerfile contienen todavía la gramática antigua que utiliza la palabra clave dedicada `MAINTAINER`, pero esto está quedando obsoleto frente al uso de las etiquetas, porque al añadirlas se pueden leer utilizando el comando `docker inspect`.

Ahora entramos en la parte funcional de `Dockerfile`, con un primer comando que permite recuperar el archivo `heartbeat.sh` creado anteriormente, por lo tanto perteneciente al contexto de compilación y copiarlo en la imagen con el nombre `entrypoint.sh` a nivel de la raíz:

```
COPY heartbeat.sh /entrypoint.sh
```

El archivo `entrypoint.sh` se debe poder ejecutar en el contenedor. Es conveniente modificar sus permisos para añadir esta característica.

```
RUN chmod +x /entrypoint.sh
```

 Si se omite este comando, se recibirá un mensaje de error `Cannot start container [...]: exec:`

`"/entrypoint.sh": permission denied`, cuando se intente ejecutar un contenedor con la imagen construida.

El siguiente comando se utiliza para dar un valor de intervalo por defecto a nuestro sistema (*heartbeat* significa "latido del corazón" en inglés y normalmente se utiliza en los servidores para llamar a los informes regulares de pruebas de actividad). Utilizaremos un retardo de dos segundos entre cada mensaje, pero como mostraremos más tarde, sigue siendo posible jugar con este valor después de construir la imagen.

```
ENV HEARTBEATSTEP 2
```

Es importante explicar bien de nuevo los dos últimos comandos. `ENTRYPOINT` es el comando que va a activar el proceso principal en el contenedor, en este caso el script de escritura del mensaje:

```
ENTRYPOINT ["/entrypoint.sh"]
```

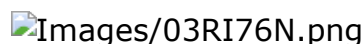
En cuanto a la palabra clave `CMD`, permite pasar el argumento por defecto al comando lanzado por `ENTRYPOINT`. En nuestro caso, queremos que el usuario de la imagen pueda pasar fácilmente el mensaje a enviar desde el comando `docker run`. Por el contrario, si no se pasa nada, mostraremos el texto *heartbeat*. Esto hace necesario declarar este argumento detrás de la palabra clave `CMD`:

```
CMD ["heartbeat"]
```

El archivo `Dockerfile` está listo y podemos situarnos ahora en el directorio que contiene los dos archivos.

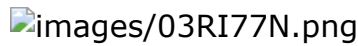
### c. Generación de la imagen

La generación de la imagen se realiza con el comando `docker build`:

Images/03RI76N.png

La visualización muestra la progresión de la compilación, según las diferentes etapas correspondientes a las líneas en el archivo `Dockerfile`. Cada etapa produce un contenedor y una imagen intermedia. Estas últimas se pueden encontrar utilizando la

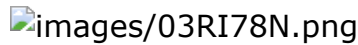
opción `-a` en el comando `docker images`, que también permite comprobar que la imagen se ha creado bien:

images/03RI77N.png


La imagen está lista y por lo tanto, vamos a poder lanzar un primer contenedor que la utiliza.

#### d. Ejecución del contenedor

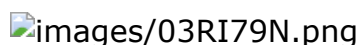
Inicialmente, se lanza un contenedor en modo interactivo y sin argumento, de manera que se pueda comprobar el comportamiento básico de la imagen:

images/03RI78N.png

El comportamiento es el esperado: en ausencia de argumentos, el mensaje `heartbeat`, aparece cada dos segundos.

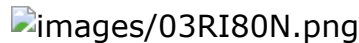
 Si obtiene un error de tipo `exec format error`, una de las posibles razones es que ha olvidado el shebang en el archivo de script (la primera línea), lo que impide que la shell sepa cómo ejecutarlo. Pero este error también aparece en muchas otras situaciones, por ejemplo, si intenta lanzar una imagen de 64 bits en una máquina host 32 bits. Si obtiene un mensaje de error como que un archivo de sistema no está presente, es posible que venga de la imagen básica que está usando. Cuando es demasiado ligera, pueden faltar los archivos necesarios para su uso. En este caso es necesario acudir a la documentación de la imagen que se está utilizando, para saber lo que hay (y garantizarlo) y lo que falta.

Atención, la combinación `[Ctrl] C` no detendrá el funcionamiento del script. Para retomar el control sobre la línea de comandos inicial, será necesario escribir la combinación `[Ctrl] P`, `[Ctrl] Q` (hablamos de operación de separación del contenedor). Llamando al comando `docker ps` después de esta operación, se puede ver que el contenedor todavía está activo:

images/03RI79N.png

Esta constatación se corrobora llamando al comando `docker logs`, que muestra los mensajes después de que hayamos salido

de la ejecución interactiva del script (se ejecuta siempre en el contenedor):



El comando `docker attach [identificador del contenedor]` se puede utilizar para conectarse al contenedor. Este comando realiza la operación inversa que la combinación de teclas mencionada anteriormente y permite volver a la consola asociada a la ejecución del contenedor.

## e. Detener y volver a lanzar el contenedor

El contenedor se puede detener por medio del siguiente comando:

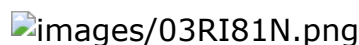
```
docker stop -t 1 repeater
```

Podemos autorizar un intervalo de tiempo muy corto entre el envío de `SIGTERM` y de `SIGKILL`, porque el proceso no tiene nada que interrumpir. En la práctica, será necesaria la segunda señal para detener el contenedor.

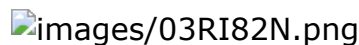
Por supuesto es posible volver a arrancar después:

```
docker start repeater
```

Este tipo de operación es visible en el log:

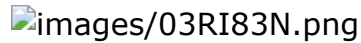


Los comandos `pause` y `unpause` permiten conseguir un resultado equivalente de manera menos brusca:



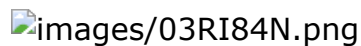
## f. Gestión de los argumentos


Para las pruebas de configuración, vamos a lanzar la imagen en modo desatendido, lo que se corresponde más con la lógica de nuestro ejemplo. Por supuesto, en este caso, no se muestra nada en la línea de comandos y solo solicitando el log de Docker podremos comprobar que el contenedor funciona correctamente:



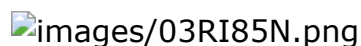
El comando `docker rm` permite eliminar el contenedor que se había lanzado previamente, la opción `-f` permite forzar su parada, necesaria para la eliminación y la opción `-v` que permite eliminar los eventuales volúmenes de datos (en este caso no hay, pero para las pruebas es una opción normal, mientras que no se debe utilizar en un entorno de desarrollo).

En el modo de funcionamiento explicado para las ejecuciones posteriores, podemos intentar añadir un argumento al comando `docker run` para comprobar que ha funcionado correctamente, cambiando el valor proporcionado por la palabra clave `CMD` de `Dockerfile` y finalmente transmitido al script apuntado por `ENTRYPOINT`, lo que a su vez se utiliza para evaluar el mensaje a enviar:



 Observe el uso de comillas para pasar un único argumento con espacios. También sería posible modificar el archivo de script utilizando `$@` en lugar de `$1`, lo que hubiera tenido por efecto utilizar para la generación de mensajes todos los argumentos pasados (separados por un espacio), en lugar de utilizar el primero.

El otro punto de variación era el intervalo de envío de los mensajes. Habíamos considerado que este modo de configuración era más adaptado al uso de una variable de entorno con un valor por defecto, establecido a dos segundos con la palabra clave `ENV` en `Dockerfile`, pero que podemos modificar con la opción `--env` durante la ejecución del contenedor con el comando `docker run`:



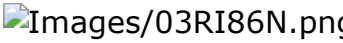
## **g. Reconstrucción de la imagen y cacheado**

La modificación de la variable de entorno, es la ocasión para presentar las capacidades de mejora de la fortaleza de funcionamiento ofrecidas por los contenedores Docker. En efecto, es posible modificar la variable de entorno y al mismo tiempo, el

valor por defecto está garantizado por `Dockerfile`. Ahora es posible simplificar el script `heartbeat.sh`, retirando las primeras líneas cuyo objetivo era prevenir la ausencia de la variable de entorno `HEARTBEATSTEP`, porque a priori nunca servirán en el ámbito de la imagen Docker. Por lo tanto, el script se puede reducir a:


```
#!/bin/bash
while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$HEARTBEATSTEP";
done
```

La ejecución del comando `build` permite volver a crear la imagen con esta nueva versión aligerada del script:

Images/03RI86N.png

Aprovecharemos esta segunda `build` para mostrar una particularidad de Docker, a saber, la presencia en un segundo paso del mensaje "Using cache". Como el archivo `heartbeat.sh` se ha modificado, el comando `COPY` del `Dockerfile` no puede utilizar la caché, así como todas las siguientes. Pero todas las anteriores se beneficiarán de un mecanismo de caché implementado por Docker, para evita volver a lanzar operaciones largas e idénticas, de una operación de `build` a otra. En nuestro ejemplo, solo está afectada una etapa sencilla pero en la práctica, esta caché permite ganar mucho tiempo. Si el funcionamiento lo necesita, es posible evitar esta caché con la opción `--no-cache`. Volveremos sobre este caso y la mejor manera de tratarlo en la sección Buenas prácticas, un poco más adelante en este capítulo.

Este mecanismo se tiene que unir con las imágenes intermedias que hemos mostrado anteriormente, durante la primera operación de creación de la imagen. Encontramos los mismos identificadores de contenedores intermedios (`2d696327ab2e` y `26282e955ad6`) para las etapas 1 y 2, durante la segunda `build`, mientras que las siguientes etapas generan otros contenedores intermedios. Son los mismos identificadores que encontramos en un comando `docker images -a`, la primera se corresponde con el código identificador de la imagen `ubuntu` con el tag `latest`.

 Respecto a las imágenes intermedias, es posible guardarlas después de una compilación correcta, usando la opción `--`

`rm=false` de `docker build`. Verá tantos contenedores como etapas de build si ejecuta el comando `docker ps -a`.

Los diferentes comandos en `Dockerfile`, no tienen el mismo comportamiento respecto a la caché. La documentación de Docker explica más en detalle las condiciones de activación de la caché para cada uno de ellos.

## 6. Comandos adicionales

### a. Gestión de los archivos

En nuestra primera realización de un `Dockerfile`, hemos utilizado el comando `COPY` para integrar archivos locales en la imagen generada. También existe otro comando con una sintaxis muy parecida (un argumento para el origen, uno segundo para el destino), pero que ya no se recomienda en la práctica, salvo en algunos casos particulares. Se trata del comando `ADD`.

`ADD` realiza la misma acción que `COPY` en un `Dockerfile`, pero ofrece dos funcionalidades adicionales:

- Si el origen es un archivo de tipo `tar`, su contenido se descomprimirá en el destino.
- Si el origen es una URL, el contenido se descargará y se depositará en el destino.

Desde entonces, el lector se podría preguntar por qué este comando, que retoma todas las funcionalidades de `COPY` y las incorpora, no debería sustituir este último. La razón es que esta sobrecarga de funcionalidad está enfocada a un uso sencillo y perfectamente desprovisto de ambigüedad. Por ejemplo, ¿cómo hacer que un archivo se copie tal cual en la imagen? Y si el formato no está soportado, el hecho de que el archivo se copie como un sencillo archivo, ¿no corre el riesgo de plantear un problema difícilmente detectable durante la compilación de la imagen?

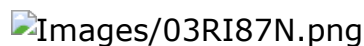
Estas razones hacen que la recomendación sea utilizar `COPY` en la mayoría de los casos y `ADD` solo para los casos particulares descritos. De paso, hemos visto que el comando `RUN` permitiría ejecutar comandos durante la compilación de la imagen, por lo que es posible llamar al ejecutable `tar` manualmente.

Respecto a la capacidad a recuperar archivos remotos, es de cualquier manera una mala práctica utilizar `ADD` para recuperar los archivos, porque estos forman parte integrante de la imagen y no se pueden eliminar, incluso si su uso es temporal. Es importante guardar las imágenes lo más ligeras posible. Excepto en casos particulares, se recomienda utilizar de nuevo el comando `RUN` para ejecutar una herramienta como `curl` o `wget`. Esto permite recuperar archivos en línea, explotarlos y - si la operación lo permite - eliminarlos. Todo se puede realizar en una única línea para solo afectar a una única capa, como se demuestra en la sección sobre las buenas prácticas, un poco más adelante en este capítulo.

## **b. Noción de contexto**

Enlazando con la gestión de archivos, es importante saber que la compilación de una imagen Docker se hace dentro de un contexto, que es el directorio que contiene el archivo `Dockerfile` que controla esta creación.

De esta manera, el comando `COPY` no podrá ir a buscar cualquier archivo a la máquina host, sino únicamente los que se encuentren en el mismo directorio (por supuesto, los subdirectorios son accesibles), que el archivo `Dockerfile`. La carga del contexto se indica durante la compilación de una imagen, como se muestra en la siguiente captura de pantalla.

Images/03RI87N.png

Esta particularidad le permite dar seguridad a su producción de imagen, no dando acceso a todo. También implica prestar atención al lugar donde crea el archivo `Dockerfile`, de manera que no se integren más archivos de lo necesario para el contexto de la imagen.

Volveremos más adelante en detalle sobre las buenas prácticas de almacenamiento pero de manera general, la mejor manera de gestionar sus imágenes es crear un directorio dedicado, con `Dockerfile` en el primer nivel de este directorio y todos los archivos necesarios, en subcarpetas.

## **c. Resultados de la asignación del proceso de arrancar**




En la imagen que nos ha servido de ejemplo anteriormente, hemos utilizado los operadores `ENTRYPOINT` y `CMD` del `Dockerfile`, explicando que el primero se corresponde con el proceso a ejecutar y el segundo con el argumento que se le debe pasar por defecto:

```
ENTRYPOINT ["/entrypoint.sh"]  
CMD ["heartbeat"]
```

Es importante volver sobre este punto particular de la complementariedad entre los dos operadores, porque hay mucha información contradictoria que se puede leer, si no prestamos atención a la calidad de las fuentes.

En primer lugar, vemos la diferencia entre `RUN` y `CMD/ENTRYPOINT`, porque es el punto más sencillo. La documentación oficial de Docker (de manera más concreta la sección sobre `RUN`, a saber <https://docs.docker.com/reference/builder/#run>), explica claramente que este operador ejecuta un comando que se le pasa como argumento en una nueva capa y lo valida (dentro de una operación `commit`). Por lo tanto, el comando `RUN` se ejecuta durante la compilación y aunque sus efectos sean visibles durante la ejecución de un contenedor, el comando no se ejecuta en ese momento. `RUN` no se debe utilizar para lanzar procesos que queremos ver activos, porque estos se terminarán al final de la compilación, sino solo para lanzar comandos de preparación de la imagen.

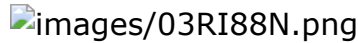
 La puesta a punto anterior puede parecer evidente (en caso contrario, ¿cuál es el interés de tener dos comandos separados entre `RUN` y `CMD/ENTRYPOINT`?), pero como algunos artículos en Internet confunden los dos, parece necesario aclarar este punto.

Para volver sobre la dualidad de `CMD` y `ENTRYPOINT` propiamente dicha, es necesario explicar que la primera se puede utilizar sin la segunda. El ejecutable que se debe arrancar durante la ejecución del contenedor, se especificará en `CMD`. El `Dockerfile` se presenta como sigue:


```
FROM ubuntu:latest  
LABEL maintainer="jp.gouigoux@free.es"  
COPY heartbeat.sh /entrypoint.sh  
RUN chmod +x /entrypoint.sh
```

```
ENV HEARTBEATSTEP 2
CMD ["/entrypoint.sh", "heartbeat"]
```

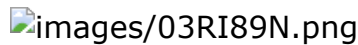
En este caso, es posible lanzar un contenedor sin argumento con el comportamiento esperado, pero la sobrecarga ya no se puede hacer sobre el único mensaje a pasar al script, como se muestra a continuación:

images/03RI88N.png

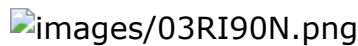
En efecto, Docker considera que la sobrecarga sustituye toda la tabla JSON (la sintaxis `["...", "..."]`) con los dos argumentos y la única palabra `"coucou"` no es un comando, de ahí el error.


 Atención, el hecho de que aparezca un error no significa necesariamente que el contenedor no se haya podido crear y en el caso anterior, será necesario eliminar el contenedor llamado `"repeater"` antes de poder crear otro con el mismo nombre.

Una manera posible de lanzar la sobrecarga es utilizar la siguiente gramática, que sustituye todo el comando a ejecutar, con el argumento asociado:

images/03RI89N.png

También es posible pasar un argumento para `entrypoint`, para regresar a la misma funcionalidad que al inicio, con la única diferencia de que el punto de entrada se pasó durante la puesta en marcha, en lugar de integrarlo en el `Dockerfile`:

images/03RI90N.png

 Aunque sea potencialmente útil esta gramática no se recomienda, porque se aleja de la intención inicial de la separación entre `ENTRYPOINT` y `CMD`, el primero conteniendo lo que no cambia de una ejecución a otra del contenedor, y el segundo siendo el operador precisamente dedicado a proporcionar un valor por defecto, destinado a intercambiarse por un valor de argumento proporcionado durante la ejecución. En los casos en los que una imagen deba poder cambiar de proceso, se recomienda crear un script de ejecución que reaccione al paso de un argumento y lance él mismo el proceso

correcto, como se ha mostrado anteriormente en el ejemplo sobre MongoDB.


#### d. Observación sobre el formato línea de comandos o ejecución

Además de la sintaxis que pasa una tabla JSON, el operador CMD permite una segunda gramática que consiste en dar un texto sencillo. Se trata de la forma "línea de comandos" (shell), en oposición a la forma "ejecución" (exec).

Supongamos que `Dockerfile` utiliza la siguiente forma, con una tabla JSON de argumentos, como se ha utilizado hasta ahora:

```
FROM ubuntu:latest
LABEL maintainer="jp.gouigoux@free.es"
COPY heartbeat.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV HEARTBEATSTEP 2
CMD ["ps", "-aux"]
```


La ejecución del contenedor muestra que el proceso principal es `ps`:


 `images/03RI91N.png`

Por el contrario, si utilizamos la forma llamada "shell" del operador CMD, el `Dockerfile` se modificará como sigue:

```
FROM ubuntu:latest
LABEL maintainer="jp.gouigoux@free.es"
COPY heartbeat.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV HEARTBEATSTEP 2
CMD ps -aux
```

Y el resultado mostrará que se ha lanzado una shell para llevar el comando:

 `images/03RI92N.png`

 Como se puede ver, se trata de una shell diferente de la lanzado durante la ejecución del archivo `entrypoint.sh` de nuestro ejemplo, que especifica por la shebang (la primera línea), que se debe ejecutar por la shell Bash.

Como le sucede a `CMD`, `ENTRYPOINT` tiene dos formas `exec` y `shell`, lo que es lógico porque solo se trata de dos operadores que al final, permiten componer una única y misma lógica de ejecución de proceso. En su forma `shell` (se pasa un sencillo texto como argumento en lugar de una tabla en formato JSON). De nuevo, se utilizará `/bin/sh -c` para lanzar el argumento de `ENTRYPOINT` y los argumentos adicionales (que provienen del `CMD` de `Dockerfile` o de la línea de ejecución `docker run`), serán ignorados.

La forma `shell` tiene la ventaja de que se sustituirán las variables como `$HOME` u otras pero por el contrario, tiene el inconveniente de que las señales no se devolverán correctamente, porque el valor que se pasa en `ENTRYPOINT` es un subcomando del proceso de PID 1, que es el shell `/bin/sh`. Con el objetivo de controlar de manera más fina este funcionamiento, acudimos a la documentación oficial (accesible en <https://docs.docker.com/reference/builder/#entrypoint>). Como regla general, lo más sencillo siempre es utilizar la forma "ejecución", que es la recomendada.

Siempre en el dominio de la operación de las señales, pero esta vez en el script `entrypoint.sh` que se lanza (o quizás en `heartbeat.sh`, que es el archivo que se volverá a copiar en la imagen con el nombre `entrypoint.sh`), sería necesaria una línea adicional para aceptar la señal `SIGTERM` enviada por la combinación `[Ctrl] C` en modo interactivo, a saber, la tercera en el script actualizado y que se presenta a continuación:

```
#!/bin/bash
set -e
trap "echo SIGNAL" HUP INT QUIT KILL TERM

while true;
do
    echo $1 \($(date +%H:%M:%S)\);
    sleep "$HEARTBEATSTEP";
done
```

La operación más sencilla para ver si su imagen soporta devolver señales, es ejecutar un `docker stop` sobre un contenedor arrancado. Si es necesario el periodo de gracia (por defecto 10 segundos) para que el contenedor se detenga, es que la señal a priori no se trata correctamente y en ese caso toma el relevo la señal `SIGKILL`, más brusca. En función del uso de su imagen, esto no es necesariamente una catástrofe, pero es una buena práctica hacer que los contenedores reasignen de manera limpia

a un SIGTERM, emulando la tarea actual y limpiando lo que sea necesario, y después detenerse antes de que se fuerce la parada.

## e. Comandos variados

WORKDIR se utiliza para fijar el directorio actual en el que está el archivo Dockerfile. Cuando se deben realizar varias operaciones encadenadas de tipo RUN, COPY o ADD, es práctico fijar el directorio de trabajo, lo que permite no tener que rescribir en cada línea. WORKDIR se puede utilizar varias veces en el Dockerfile. El directorio de trabajo evoluciona a medida que avanza la compilación.

LABEL permite ofrecer parejas clave/valor, que servirán de metadatos que describen la imagen cuando se difunde y utiliza. Esta técnica permite comunicar fácilmente información sencilla a los usuarios de la imagen, ya sea de su propio creador o de otras personas. Un uso clásico es asociar la versión de la imagen en una etiqueta:

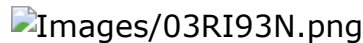
```
LABEL version="1.0"
```

Los metadatos también son un lugar particularmente apropiado para contener instrucciones de uso:

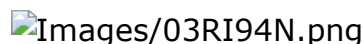
```
LABEL description="This image emits a regular message on STDIN"
LABEL readme="The HEARTBEATSTEP environment variable shall be \
modified with a value in seconds in order to change the time \
interval between messages, while messages themselves can be \
modified by passing the updated value as a parameter placed \
at the end of the docker run command line."
```

El símbolo \ permite continuar una entrada en la línea siguiente. Por convención, se precede por un espacio. Volveremos sobre su uso en las buenas prácticas más adelante.

El comando `docker inspect` permite encontrar los metadatos en una imagen dada (no olvide compilar justo antes):

Images/03RI93N.png

Las etiquetas también se encuentran en los contenedores lanzados desde esta imagen:

Images/03RI94N.png

USER es la palabra clave del Dockerfile que permite modificar el usuario actual para los siguientes comandos de la operación de creación de la imagen y de hecho, para los contenedores que se lanzarán a partir de la imagen creada. Ya que las imágenes se encargan de manera tradicional de la ejecución de un único proceso, y que la única vía de comunicación que se le asigna es, en general, un puerto, muchos Dockerfile no especifican este comando lo que hace que el proceso no funcione en modo "menos privilegiado". En un enfoque defensivo, es una buena práctica asignar un usuario dedicado con permisos reducidos, lo que reduce el impacto de un eventual fallo. La sintaxis de este comando es muy sencilla (el grupo es opcional):


## **Modificar el usuario actual**

```
USER usuario[:grupo]
```

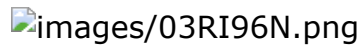
ARG es un comando que también hay que explicar, solo porque es fácil perderse entre los diferentes medios de pasar valores de argumentos a una imagen. Hemos visto que era posible pasar una variable de entorno del host al contenedor, cuando este se lanza. También es posible pasar diferentes comandos, sobrecargando por la línea de comandos el contenido fijado por defecto por el comando CMD. El comando ARG permite pasar valores durante la compilación de la imagen. Para esto, es necesario declarar la variable en el Dockerfile con un valor por defecto. En nuestro ejemplo, podemos utilizar ARG para modificar durante la compilación el valor por defecto de HEARTBEATSTEP que va en la imagen (por supuesto, sin impedir que el valor de HEARTBEATSTEP se pueda modificar, porque está ubicado en una variable de entorno). El archivo Dockerfile se parecía al siguiente:

```
FROM ubuntu:latest
ARG HEARTBEATSTEPDEFAULT=2
LABEL maintainer="jp.gouigoux@free.es"
COPY heartbeat.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV HEARTBEATSTEP ${HEARTBEATSTEPDEFAULT}
ENTRYPOINT ["/entrypoint.sh"]
CMD ["heartbeat"]
```

A continuación, es el momento de intervenir durante la compilación para cambiar este valor por defecto:

 Images/03RI95N.png

Como se explica más atrás, esto no tiene nada que ver con el hecho de que sea posible modificar el valor de `HEARTBEATSTEP` en la ejecución:



El uso típico de los argumentos de build, es permitir generar varias imágenes con comportamientos diferentes, sin duplicar los archivos `Dockerfile`. De esta manera, podríamos obtener imágenes como la siguiente:

```
docker build -t jpgougoux/repeater-quick --build-arg
HEARTBEATSTEPDEFAULT=1 .

docker build -t jpgougoux/repeater-slow --build-arg
HEARTBEATSTEPDEFAULT=10 .
```

Este ejemplo cierra nuestro estudio sobre los operadores que se pueden usar en el archivo `Dockerfile`. Una vez más, el objetivo de este libro no es explicar de manera exhaustiva las funcionalidades de Docker, sino mostrar que la gramática de los `Dockerfile` es muy sencilla y que por el momento, no contiene realmente otros operadores que se puedan considerar imprescindibles en este estado. El comando `SHELL` se utiliza sobre todo para pasar del shell `CMD` o `PowerShell` en las imágenes `Windows`, que no se abordan en este libro. Los comandos `ONBUILD`, `HEALTHCHECK`, `EXPOSE` y `VOLUME` se detallarán a continuación.

# Compartición y reutilización sencilla de imágenes

## 1. Enviar a su cuenta Docker Hub

En el capítulo anterior, hemos mostrado cómo conectarse a Docker Hub y asociar la cuenta con un almacén en GitHub. A continuación, nos aseguramos de que este almacén cree una imagen Docker actualizada en cada `commit` de código fuente.

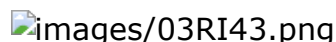
También es posible ubicar en un registro Docker una imagen creada localmente, como las que hemos implantado en este capítulo. Para esto, utilizaremos un comando que todavía no hemos abordado.

### Enviar una imagen local al registro Docker Hub

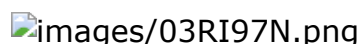
```
docker push [imagen]
```

Para que este comando funcione, es necesario que la imagen tenga un nombre completo, usando como prefijo el nombre de la cuenta Docker Hub. Esta es la razón por la que la imagen creada más atrás, se llamó `jpgouigoux/repeater`. También hubiera sido posible llamar simplemente `repeater` ya que el ejercicio era local, y después utilizar el comando `docker tag` para añadirle una etiqueta adicional correspondiente a este nombre completo.

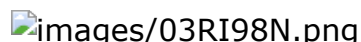
Apliquemos este comando a la imagen que hemos creado anteriormente:



Después de algunos minutos, en función del tamaño de la imagen y de la rapidez de la conexión Internet utilizada, la imagen se puede ver en Docker Hub:




Haciendo clic en el almacén, puede acceder a sus propiedades:







Por lo tanto, es posible hacer que esta imagen sea privada (solo la cuenta utilizada para desplegarla la podría ver y recuperar), utilizando el comando `Make Private` en la pestaña `Settings`. En una cuenta gratuita, Docker Hub solo permite por el momento una única imagen privada. Si desea utilizar Docker Hub de manera más intensiva sin hacer sus imágenes privadas, una solución es suscribirse a una oferta comercial (ver <https://registry.hub.docker.com/plans/>). Otra es mantener su propio registro.

 Veremos un poco más adelante que Docker Hub no es el único medio para hacer disponibles sus imágenes. En un primer enfoque, se trata del método más sencillo: una conexión es suficiente.

Por supuesto, el tipo de envío realizado por el comando `push`, solo se ocupa de la imagen propiamente dicha. Sin código fuente ni `readme.md`, etc. Es posible editar directamente una descripción en la página que se ha mostrado justo por encima, haciendo clic en el icono que representa un lápiz, en la parte superior derecha de la descripción:

 `images/03RI99N.png`

Una interfaz integrada permite dar una descripción resumida y la misma operación está disponible en el panel de la descripción larga de la imagen:

 `images/03RI100N.png`

Atención, si interrumpe voluntariamente el proceso de envío de una imagen en el registro, es posible que durante el siguiente intento reciba un error de tipo `push is already in progress` porque Docker anula el comando, pero no la descarga en segundo plano. Puede esperar a que la descarga termine para volver a comenzar o arrancar el demonio para anularla. El comando a utilizar es `sudo servicio docker restart` en Ubuntu.

## 2. Exportación e importación en forma de archivos


Algunas veces no es posible confiar en el registro Docker Hub para pasar las imágenes de una máquina a otra o distribuirlas a

personas autorizadas, por razones de confidencialidad o propiedad intelectual. Una forma, aunque sea más manual, es exportar una imagen como un archivo. Esto permite un transporte por cualquier medio, del más sencillo (copia en una memoria USB) al más sofisticado (intercambio por contraseña digital).

### Exportar una imagen en forma de archivo

```
docker save -o [archivo] [imagen]
```

Una buena práctica es no situar el archivo en el mismo lugar que el archivo `Dockerfile`, para que no pueda contaminar el contexto durante una próxima operación de compilación:

 images/03RI101N.png

Por supuesto, este comando solo tiene interés acompañado de su contraparte.

### Importación de un archivo de imagen


```
docker load -i [archivo]
```

Observe que el archivo de la imagen está completo, lo que es muy práctico porque no necesita dependencias para utilizarse en cualquier máquina, pero también muy costoso en espacio en disco. En nuestro ejemplo, aun siendo muy sencillo, el archivo ocupa 125 MB: menos de 1 MB para la parte añadida por nosotros. El resto se corresponde con la imagen `ubuntu` sobre la que nos hemos basado y que forma parte del archivo.

## Buenas prácticas

### 1. Principio de la caché de las imágenes

Un usuario atento habrá observado la diferencia de tiempo entre la primera ejecución de un contenedor y las siguientes. La diferencia es particularmente notable cuando un contenedor se basa en una imagen `ubuntu` o `debian`. Durante la primera ejecución, Docker descarga alrededor de 125 MB para la primera y 100 MB para la segunda. Durante la segunda ejecución así como para las siguientes, esta descarga ya no tiene lugar y la puesta en marcha del contenedor es mucho más rápida.

 Hablamos de la puesta en marcha del contenedor, pero también se puede tratar del único comando `pull` de recuperación de la imagen desde el registro. Se observa que el comando `run` normalmente se utiliza más y de todas formas se lanza un comando `pull`, si la imagen no está disponible localmente.

Docker gestiona una caché para las imágenes. Esta caché es local y varias veces se ha observado su contenido con el comando `docker images`. La caché simplemente es lo que hemos llamado hasta aquí la lista de las imágenes locales. Un comando `docker pull` solo lleva a local (por lo tanto, la cachea) una imagen que inicialmente existía en el registro remoto.

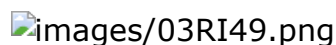
Como hemos visto en el capítulo sobre los aspectos principales de Docker, instanciar un contenedor vuelve a crear, desde el punto de vista del contenido, una nueva capa en modo escritura sobre una pila de capas existentes, que solo son accesibles en modo lectura. En la práctica, estas capas están sistemáticamente presentes durante la ejecución, pero por razones diferentes en función del contexto:

- Si la imagen se ha construido localmente, el comando de compilación `docker build` ha recuperado automáticamente toda la pila de imágenes sobre la que se basaba `Dockerfile`, con la palabra clave `FROM`.

- Si la imagen se ha recuperado desde un registro, el comando `pull` o `run` ha traído todas las capas necesarias y no solo la más alta, correspondientes al tag proporcionado. Por supuesto, el mecanismo de caché se aplica a todos los niveles y si una imagen se basa en otra, que a su vez se basa en una imagen ya presente, solo se recuperan las dos primeras.
- Si la imagen se ha cargado desde un archivo por el método `load`, las imágenes de las que también depende se archivan localmente si no existían, porque el archivo contiene todo.

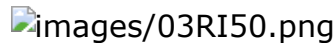
La cuestión sobre el borrado de una imagen por parte de otra que hubiera tenido el mismo tag en un momento dado, ya no se plantea porque prevalece el identificador hexadecimal de una imagen. Imaginemos por ejemplo que se ha creado un archivo con una imagen que se basa en `ubuntu:latest`, pero hace de esto varios meses. ¿Qué sucede si no cargamos el archivo en la caché local y la versión `ubuntu:latest` ha cambiado desde entonces? De hecho, nada especial, porque la imagen cargada depende de una imagen Ubuntu con un identificador y el resto de imágenes locales pueden depender de otra imagen Ubuntu. Los identificadores no son los mismos y la imagen cargada sabe que siempre se debe basar en la imagen a partir de la que ha sido compilada (y que era la más actualizada en aquel momento). El tag `latest` se aplica a otra imagen, con un identificador diferente, pero esto no cambia nada para la imagen cargada. Sin embargo, si se compila de nuevo la imagen a partir de su `Dockerfile`, se actualizará para apuntar a la nueva imagen, porque el tag `latest` se utiliza en `Dockerfile` al que apunta.

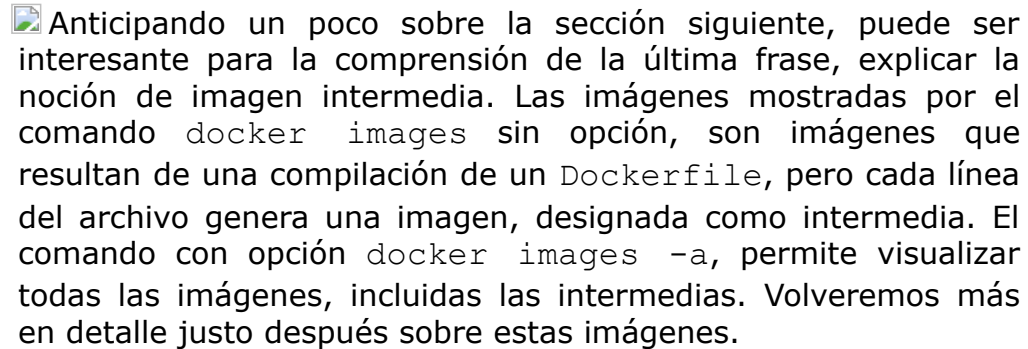
Lo más sencillo es mostrar esta evolución por medio de algunos esquemas, empezando por describir las imágenes en la caché local alrededor de mediados de marzo del 2015, es decir, cuando la imagen que nos servirá de ejemplo se crea por primera vez, como fruto de la primera edición del presente libro. Ahora se basa en la que en ese momento era la versión Long Term Support más reciente (`latest`, en inglés) de la imagen `ubuntu`, es decir, la versión 14.04:



Se ha utilizado el comando `docker save` para crear un archivo correspondiente a la imagen etiquetada `imagetest`. El comando incluye toda la pila de imágenes dependientes, el archivo agrupa las imágenes `e5afb9797f1a` y `d0955f21bf24` (así como todas

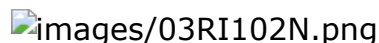
las imágenes del componente ubuntu, pero para simplificar, no mostramos las imágenes intermedias):

images/03RI50.png

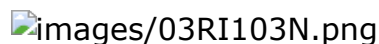
Anticipando un poco sobre la sección siguiente, puede ser interesante para la comprensión de la última frase, explicar la noción de imagen intermedia. Las imágenes mostradas por el comando `docker images` sin opción, son imágenes que resultan de una compilación de un `Dockerfile`, pero cada línea del archivo genera una imagen, designada como intermedia. El comando con opción `docker images -a`, permite visualizar todas las imágenes, incluidas las intermedias. Volveremos más en detalle justo después sobre estas imágenes.

Dos años más tarde, durante la escritura de la segunda edición de este mismo libro (versión que usted está leyendo), la versión LTS más actual de la imagen `ubuntu` ya no es la misma: se trata de la 16.04, llamada Xenial Xerus.

El siguiente comando permite encontrar su identificador, la única información estable, las etiquetas se podían - como se acaba de mostrar - mover de una imagen a otra, prestando atención solo al editor garantizando que estos movimientos tienen sentido:

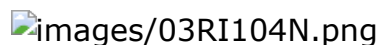
images/03RI102N.png

Otra máquina candidata a recibir el archivo generado anteriormente, podría presentar la siguiente estructura:

images/03RI103N.png

Vemos claramente que el identificador de la imagen ya no es el mismo: hay una nueva imagen que ha aparecido y la etiqueta (tag en lenguaje Docker) `ubuntu:latest` apunta a esta nueva imagen.

Si se usa un comando `docker load` para restaurar la imagen archivada en esta máquina destino, obtendremos el siguiente contenido de la caché local de las imágenes:

images/03RI104N.png

La imagen correspondiente a la antigua versión de Ubuntu (aquella etiquetada como `latest` durante la compilación de `image-test`), se ha restaurado pero sin su etiqueta. Esto muestra la diferencia clara que hay entre lo que normalmente se llama el "nombre" de una imagen y su identificador.


El "nombre" es la composición textual entre el almacén (`ubuntu`) y el tag (`latest`, por ejemplo) y no se trata realmente de una etiqueta dada a una entidad, sino de una descripción que puede ser múltiple. Por ejemplo, durante la escritura de este capítulo, a la misma imagen del almacén `ubuntu` se le añadieron las siguientes etiquetas:

- `16.04`: la versión según su expresión corta.
- `xenial-20170915`: la versión con la fecha de construcción (15 de septiembre del 2017) y la serie (Xenial Xerus es el nombre del código de la versión 16.04).
- `xenial`: el nombre del código solo, lo que permite apuntar a la última versión conocida de Xenial Xerus, una 16.04 la más actualizada posible.
- `latest`: en la actualidad, la etiqueta `latest` se asigna a esta versión 16.04, porque se trata de una versión Long Term Support. Hay otras versiones más recientes de Ubuntu, a saber, la 17.04 y la 17.10, pero Ubuntu saca cada tres años una versión con un soporte a largo plazo y ha decidido asignar el tag `latest` a esta, lo que obedece a una lógica de estabilidad.

El identificador de una imagen se puede ver como la huella de una imagen: tan pronto como el contenido cambia, el identificador debe cambiar. De ahí su representación en forma de *hash*.

Anteriormente hablamos de estabilidad y es esta cualidad primordial de los enfoques por contenedores, lo que está detrás del funcionamiento explicado anteriormente. Cuando se restaura la imagen archivada, es fundamental que toda la pila de imágenes sobre la que se basa sea absolutamente idéntica a la original. Una de las promesas del enfoque por contenedor es garantizar que una imagen funcionará siempre de la misma manera, de una máquina a otra, de un contexto de uso a otro y de una ejecución a otra. Por lo tanto es crucial que no pueda intervenir ningún cambio que pueda entorpecer esta estabilidad. Esta es también la razón por la que un archivo de imagen realizado por `docker save`, incorpora todo. Durante la restauración, si ya existe una imagen con el mismo identificador exactamente, tanto mejor. Pero si ha

cambiado aunque sea un solo byte, se restaurará la imagen original.

 Entender la diferencia entre un tag y un identificador de imagen es absolutamente fundamental para la correcta implantación de Docker.


## 2. Principio de la caché durante la compilación

### a. Vuelta a las imágenes intermedias

Hemos mencionado brevemente un poco más atrás la noción de imagen intermedia. Conviene ahora explicar más en detalle con qué se corresponde. Por ejemplo, tomemos la imagen `repeater` que ya nos ha servido y volvamos a definir el `Dockerfile` como sigue:

```
FROM ubuntu:latest
LABEL maintainer="jp.gouigoux@free.es"
LABEL version="1.0"
LABEL description="This image emits a regular message on STDIN"
COPY heartbeat.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV HEARTBEATSTEP 2
ENTRYPOINT ["/entrypoint.sh"]
CMD ["heartbeat"]
```

Si compilamos este `Dockerfile`, obtenemos la imagen correspondiente con el nombre elegido, en nuestro caso `repeater`:

 Images/03RI105N.png

Para profundizar en el análisis de lo que sucede en el comando, vamos a recurrir a un comando que permite visualizar la arborescencia de las imágenes. Este comando ha formado parte en el pasado de `docker images`, como una opción `--tree` que ya no se usa. En su lugar, la página <https://github.com/moby/moby/issues/1776> recomienda el uso de un comando externo, difundido en forma de un contenedor:

```
docker run --rm -v /var/run/docker.sock:/var/run/docker.sock
nate/dockviz images -t
```

El hecho de que el comando no forme parte más de la herramienta Docker en sí misma y se base en un contenedor externo, planteará problemas en producción, independientemente de la confianza que podamos tener en el propietario de la cuenta `nate`. El hecho de utilizar un volumen para dar a este contenedor acceso al socket que permite controlar el demonio Docker de la máquina alojada, también necesita un grado de confianza que puede que no tenga en su máquina local. En este caso, lo más sencillo es crear una máquina virtual o una máquina en el cloud, que pueda eliminar después de algunas operaciones. La controversia ha estado alrededor de Docker para volver a restablecer este comando, pero por el momento la empresa no ha querido, argumentando que quiere que esta herramienta siga siendo externa. Es cierto que mantener ligera una arquitectura es un buen argumento, pero no resuelve la necesidad de confianza en los comandos que son necesarios para el correcto mantenimiento de una infraestructura.

Planteadas estas precauciones, la ejecución del comando permite visualizar la arborescencia completa de la imagen de nuestro ejemplo:

```
<missing> Virtual Size: 121.6 MB
└─<missing> Virtual Size: 121.6 MB
  └─<missing> Virtual Size: 121.6 MB
    └─<missing> Virtual Size: 121.6 MB
      └─<missing> Virtual Size: 121.6 MB
        └─2d696327ab2e Virtual Size: 121.6 MB Tags: ubuntu:latest
          └─da18278b451c Virtual Size: 121.6 MB
            └─4ad67728ee5f Virtual Size: 121.6 MB
              └─bda976cf6782 Virtual Size: 121.6 MB
                └─f4179fe6499c Virtual Size: 121.6 MB
                  └─0c70edab641b Virtual Size: 121.6 MB
                    └─4ebb970f936e Virtual Size: 121.6 MB
                      └─dbf9a4777376 Virtual Size: 121.6 MB
                        └─121953a0103e Virtual Size: 121.6 MB
Tags: repeater:latest
```

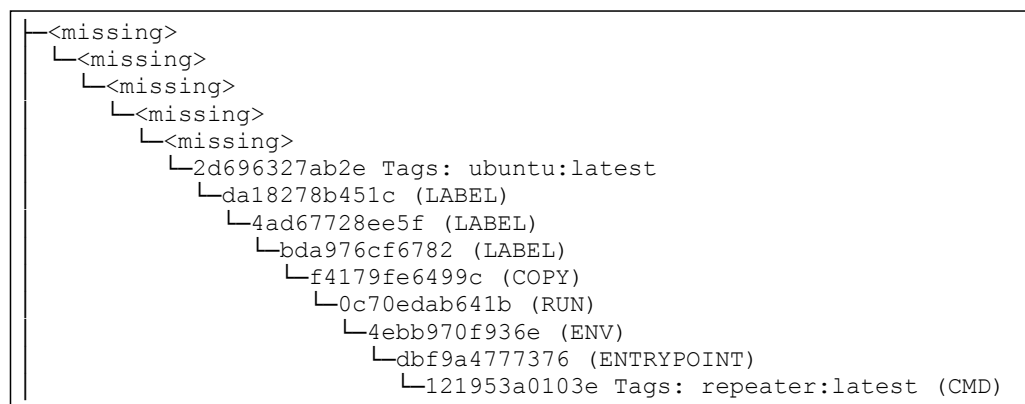
Con esta visualización se puede aprender mucho. En la parte superior de la arborescencia, están las imágenes intermedias y esto, al sexto nivel, se corresponde con la imagen `ubuntu` en su tag `latest`, en la que habíamos basado la imagen `repeater`. Desde la séptima, se encuentran ocho niveles de imágenes, siete intermedios y la última correspondiente a la imagen `repeater`.


Este número se corresponde con el número de etapas del archivo `Dockerfile`. Este último incluye una línea `FROM` (que explica por qué la arborescencia se une en `2d696327ab2e`, que es el identificador de la imagen `ubuntu` etiquetada `latest` durante la compilación, como habíamos visto anteriormente), así como ocho



comandos LABEL (tres veces), COPY, RUN, ENV, ENTRYPOINT y para terminar CMD, que cierra la lista y se corresponde con la imagen final llamada `repeater` (y etiquetada por defecto `latest`, si no especificamos un tag durante la compilación).

Eliminando el tamaño virtual para facilitar la visualización y añadiendo entre paréntesis las etapas sucesivas de Dockerfile, podemos volver a escribir la salida anterior como se muestra a continuación:



 A propósito del tamaño virtual, se corresponde con el tamaño de la imagen acompañada de todas las imágenes de las que depende. Por lo tanto, es lógico que no haga más que aumentar a medida que se desciende en la lista. Comprobamos además que en nuestro ejemplo no aumenta casi nada (un salto de 200 KB en la segunda etapa y después nada), lo que prueba que las últimas etapas de creación de la imagen Ubuntu, así como los comandos de construcción de la imagen `repeater`, ocupan muy poco espacio en las imágenes generadas.

## b. Anatomía de una compilación de imagen

La razón por la que hablamos de las imágenes intermedias, es explicar la potencia de la caché. En la sección anterior, hemos mostrado que Docker implementaba una caché para evitar recuperar por ejemplo, la imagen `ubuntu` cada vez que se instancia un contenedor basada en ella. El mismo mecanismo se implementa para cada una de las imágenes intermedias y permite beneficiarse de un rendimiento incrementado durante la compilación de una imagen: todas las etapas que no hayan sufrido ningún cambio, no tienen ninguna razón dirigirse a una imagen intermedia diferente, y esta se recupera de la caché en lugar de reconstruirla con el comando `docker build`.

De nuevo, lo más sencillo es ilustrar el efecto de una nueva compilación sobre nuestro Dockerfile modificado. Cuando habíamos compilado la imagen repeater, habíamos obtenido la siguiente salida:

```
swarm@node1 ~/repeater $ docker build -t repeater .
Sending build context to Docker daemon 3.072kB
Step 1/9: FROM ubuntu:latest
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest:
sha256:d45655633486615d164808b724b29406cb88e23d9c40ac3aaaa2d69e79e3bd
5d
Status: Downloaded newer image for ubuntu:latest
---> 2d696327ab2e
Step 2/9: LABEL maintainer "jp.gouigoux@free.es"
---> Running in 833bedea5bcd
---> da18278b451c
Removing intermediate container 833bedea5bcd
Step 3/9: LABEL version "1.0"
---> Running in 7c290cbff49f
---> 4ad67728ee5f
Removing intermediate container 7c290cbff49f
Step 4/9: LABEL descripción "This image emits a regular message on
STDIN"
---> Running in 78813257746b
---> bda976cf6782
Removing intermediate container 78813257746b
Step 5/9: COPY heartbeat.sh /entrypoint.sh
---> f4179fe6499c
Removing intermediate container 03ecffe5ec90
Step 6/9: RUN chmod +x /entrypoint.sh
---> Running in 652f9dc5b955
---> 0c70edab641b
Removing intermediate container 652f9dc5b955
Step 7/9: ENV HEARTBEATSTEP 2
---> Running in 05cba731a0fa
---> 4ebb970f936e
Removing intermediate container 05cba731a0fa
Step 8/9: ENTRYPOINT ./entrypoint.sh
---> Running in 0212204cd9ba
---> dbf9a4777376
Removing intermediate container 0212204cd9ba
Step 9/9: CMD heartbeat
---> Running in 1954826afbf1
---> 121953a0103e
Removing intermediate container 1954826afbf1
Successfully built 121953a0103e
Successfully tagged repeater:latest
```

Esta visualización viene a confirmar el análisis de la correspondencia entre las imágenes intermedias y las etapas de Dockerfile. Nos dice además cuál es la manera que tiene Docker de proceder para compilar una imagen.

En cada paso (*step*, en inglés), Docker:

- arranca un contenedor temporal (en la etapa 2, tiene el identificador `833bedea5bcd`),
- lanza el comando sobre este contenedor,
- realiza una operación de `commit` en una imagen intermedia, que da el código (siempre en la etapa 2, es esta vez el código `da18278b451c`, que es el mismo que el que teníamos en la lista anterior como primera etapa de tipo LABEL),
- elimina el contenedor intermedio (mensaje `removing intermediate container 833bedea5bcd`, en el caso de la etapa 2),
- pasa a la etapa siguiente y reproduce el mismo proceso, basándose esta vez en un contenedor instanciado desde la imagen intermedia anterior,
- alcanzada la última etapa, Docker etiqueta la imagen final y comunica el éxito de la operación, recordando el código de la imagen generada, en nuestro caso `121953a0103e`.

### c. Análisis de una modificación de la Dockerfile

Imaginemos ahora que deseamos cambiar el intervalo de tiempo por defecto de nuestra imagen `repeater`, llevándolo solo a una segunda. Modificaremos el `Dockerfile` como sigue (solo se ve impactada la línea que empieza por `ENV`):

```
FROM ubuntu:latest
LABEL maintainer="jp.gouigoux@free.es"
LABEL version="1.0"
LABEL description="This image emits a regular message on STDIN"
COPY heartbeat.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENV HEARTBEATSTEP 1
ENTRYPOINT ["/entrypoint.sh"]
CMD ["heartbeat"]
```

El hecho de compilar la imagen generaría la siguiente visualización:

```
swarm@node1 ~/repeater $ docker build -t repeater .
Sending build context to Docker daemon 3.072kB
Step 1/9: FROM ubuntu:latest
----> 2d696327ab2e
Step 2/9: LABEL maintainer "jp.gouigoux@free.es"
----> Using cache
----> da18278b451c
```

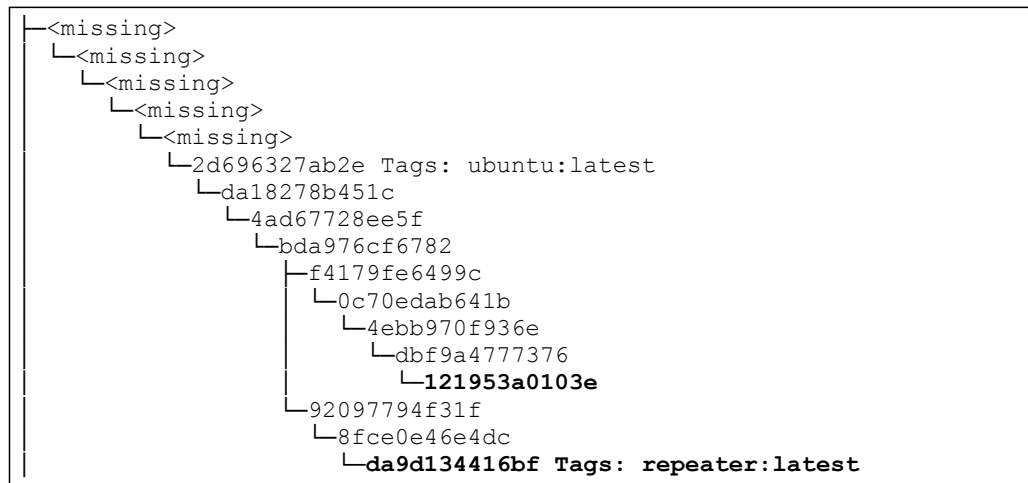
```
Step 3/9: LABEL version "1.0"
---> Using cache
---> 4ad67728ee5f
Step 4/9: LABEL description "This image emits a regular message on
STDIN"
---> Using cache
---> bda976cf6782
Step 5/9: COPY heartbeat.sh /entrypoint.sh
---> Using cache
---> f4179fe6499c
Step 6/9: RUN chmod +x /entrypoint.sh
---> Using cache
---> 0c70edab641b
Step 7/9: ENV HEARTBEATSTEP 1
---> Running in 4c49ae1e5835
---> 92097794f31f
Removing intermediate container 4c49ae1e5835
Step 8/9: ENTRYPOINT ./entrypoint.sh
---> Running in 59be56f14e16
---> 8fce0e46e4dc
Removing intermediate container 59be56f14e16
Step 9/9: CMD heartbeat
---> Running in b23ef8cf25c4
---> da9d134416bf
Removing intermediate container b23ef8cf25c4
Successfully built da9d134416bf
Successfully tagged repeater:latest
```

Si nos interesamos por la primera etapa de la compilación propiamente dicha (step 2 en la salida anterior), el step 1 se corresponde con la recuperación de la imagen básica), el mensaje de ejecución de un contenedor intermedio se sustituye por Using cache. Docker ha detectado que no ha cambiado nada y decide utilizar su caché en la imagen intermedia. Además vemos que el identificador resultante para esta etapa es el mismo, a saber, en nuestro ejemplo da18278b451c.

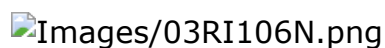
Pero durante la etapa nº7 (que se corresponde con el comando ENV que hemos modificado), la caché lógicamente ya no es utilizable y hay una recompilación real, que genera un identificador diferente al que teníamos anteriormente para esta misma etapa.

Lógicamente, todas las etapas de compilación siguientes se ven impactadas: como la imagen resultante de la etapa nº7 ha cambiado, la nº8 no puede utilizar su caché, por lo que debe recompilar su imagen que será diferente y continuar de esta manera hasta alcanzar una imagen final (etiquetadas repeater:latest) con un identificador diferente de la primera compilación, en este caso da9d134416bf en lugar de 121953a0103e, lo que indica que su comportamiento final ya no es el mismo (intervalo de un segundo en lugar de dos por defecto).

La utilización del mismo comando que el indicado anteriormente, muestra la nueva arborescencia, que se bifurca a la séptima etapa después de la imagen básica `ubuntu:latest` y con la etiqueta `repeater:latest`, en los nuevos identificadores de imágenes (la información del tamaño virtual también se ha eliminado aquí y los dos identificadores de imagen final se han resaltado en negrita para facilitar el análisis):

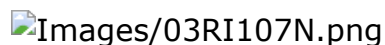


La rama de `121953a0103e` es difícilmente accesible, porque no hay ninguna etiqueta que apunte a ella. Esto se manifiesta por medio de una visualización menos elegante de un comando `docker images`:



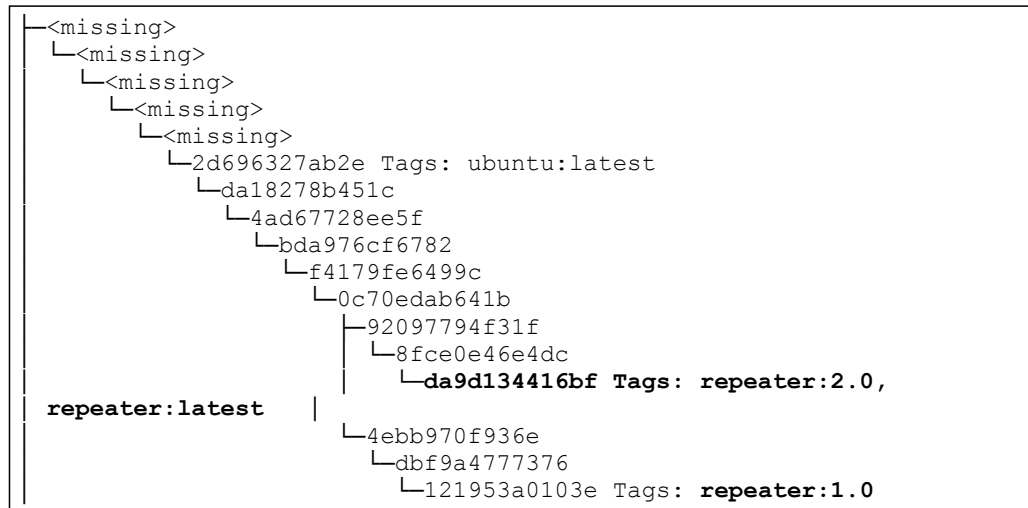
## d. Gestión correcta de las etiquetas

Esta es la razón por la que siempre se recomienda dar un número de versión explícito durante la compilación y mover a continuación la etiqueta `latest` con el comando `tag`. En nuestro caso, podemos empezar corrigiendo esta entrada antigua y dándole la versión `1.0` con el comando `docker tag 121953a0103e repeater:1.0`. El resultado será una lista de imágenes más limpia:



Y si queremos obtener un estado impecable, otro comando `docker tag repeater:latest repeater:2.0` permitirá asignar una segunda etiqueta a la imagen asociada por el

momento únicamente a `repeater:latest`. La arborescencia será idéntica a la que hubiéramos obtenido si hubiéramos compilado desde el inicio con los números de versiones correctos:





El encadenamiento recomendado de compilación de las imágenes, es el siguiente:

- `docker build -t repeater:1.0`
- `docker tag repeater:1.0 repeater:latest`
- modificación del Dockerfile
- `docker build -t repeater:1.0`
- `docker tag -f repeater:1.0 repeater:latest`


Es necesaria la opción `-f` para el último comando, porque la etiqueta `latest` ya fue asignada a la imagen etiquetada `1.0`, es necesario forzar el movimiento de la etiqueta `latest` a otra imagen del mismo almacén (`repeater`). También hubiera sido posible eliminar el tag `latest` de la imagen en versión `1.0`, en la que se utiliza el comando `rmi`, y después asignarle sin que sea necesario forzar el movimiento con la opción `-f`.

Cuando se utiliza el comando `rmi` en una etiqueta mientras que la imagen contiene otras, el mensaje resultante indica únicamente el "des-etiquetado".

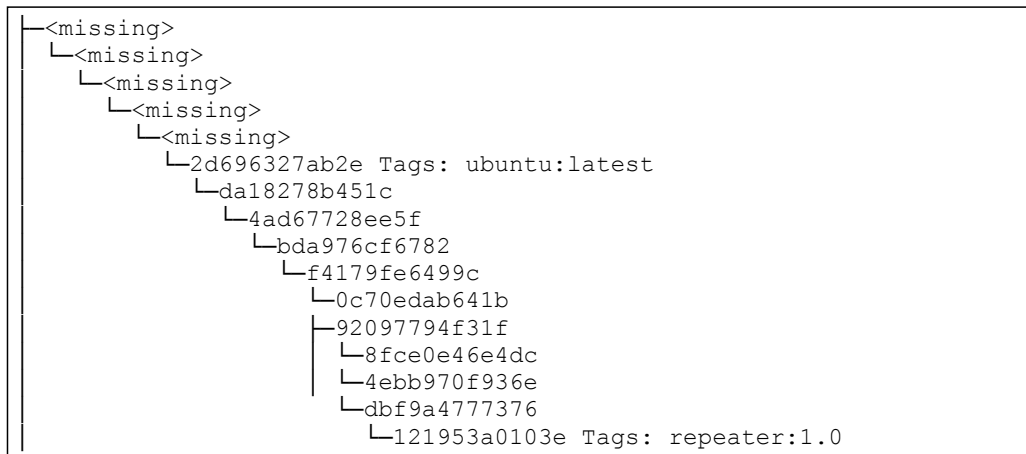
 Images/03RI108N.png

 Puede parecer extraño llamar a un comando `rmi` (abreviación de *remove image*, es decir eliminar la imagen, en inglés) mientras que la imagen sigue existiendo, pero la eliminación de la imagen y por lo tanto de todas las etiquetas, mientras que solo se ha indicado una en el comando, sería todavía más desconcertante para su uso.


Por el contrario, cuando se utiliza el comando `rmi` en la última etiqueta de una imagen, esta se elimina de manera efectiva y a menos que no se use la opción `--no-prune`, todas las imágenes intermedias también. Por ejemplo, después de haber eliminado el tag `latest` como se muestra anteriormente, también podemos eliminar el tag `repeater:2.0`, con la famosa opción:

 Images/227.png

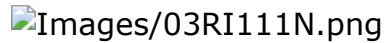
El mensaje es claro sobre el hecho de que solo se ha eliminado una única imagen (hasta ahora habíamos utilizado identificadores cortos, pero lo que se muestra se corresponde con la versión larga). La arborescencia resultante sería como la siguiente:



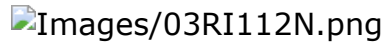
Y de nuevo, nos encontramos con una entrada no identificada en la lista de imágenes finales (las no-intermedias, es decir, aquellas que se corresponden con un nodo final en el árbol o bien han recibido una etiqueta):

 Images/03RI110N.png

Es necesario eliminar esta imagen por su identificador, para que las cosas vuelvan a estar en orden:



Para volver a cero, eliminamos el tag `repeater:1.0`, teniendo cuidado esta vez con no utilizar la opción `--no-prune` (cuya utilidad en una situación real no es evidente), lo que provoca la eliminación de las ocho imágenes intermedias, además de la imagen que se corresponde con la primera versión de nuestro ejemplo:



### **e. Invalidación de la caché por modificación de la imagen básica**

Hemos visto más atrás que el mecanismo de caché se desactiva tan pronto como se producía una modificación en el archivo `Dockerfile`. Esto es cierto para cualquier modificación, incluida la sencilla adición de un espacio, que no cambia en nada el significado del comando afectado.

La decisión de invalidar la caché (se trata de la expresión dedicada), implica algunos otros casos.

El primero aparece cuando la imagen básica apuntada por la etiqueta en el `FROM`, ya no es la misma (no se puede decir realmente que haya cambiado, porque una imagen es, como hemos visto, inmutable: solo la etiqueta se desplaza a otra imagen, lo que da la impresión de que ha cambiado). Por supuesto, todo el interés de utilizar versiones en las imágenes es estabilizar su contenido de manera que se evite impactar en las personas que las utilizan. Por lo tanto, no deberíamos nunca entregar una imagen modificada bajo un número de versión ya utilizado. Pero nada lo impide y es interesante saber que esto conduce a una recompilación de la imagen basada en la que ha evolucionado.

Este primer caso es muy explicativo sobre el funcionamiento de las imágenes. Por lo tanto, vamos a detallarlo:

Elimine todas las referencias a `repeater` si todavía hay (normalmente, el final de la sección anterior ha permitido hacer limpieza).

Sitúese en el directorio `repeater` que contiene el `Dockerfile`, utilizado como ejemplo hasta ahora.



Compile la imagen `repeater`, dándole una etiqueta `1.0`.

Cree otro directorio al mismo nivel que `repeater` (sobre todo no dentro de este mismo directorio) y sitúese dentro. En nuestro ejemplo, lo llamaremos `dependance`.

Cree un archivo `Dockerfile`.

Añada en este archivo las siguientes líneas:

```
FROM repeater:1.0
ENV MESSAGEDEFAULT heartbeat
```

Guarde el archivo.

Compile este `Dockerfile` con el nombre de imagen `dependance:1.0`.

La arborescencia de las imágenes se presenta como sigue (la información de tamaño virtual se ha eliminado sistemáticamente):

```
└─<missing>
  └─<missing>
    └─<missing>
      └─<missing>
        └─<missing>
          └─2d696327ab2e Tags: ubuntu:latest
            └─f7d11a50b838
              └─df1153da0e1c
                └─dfd6d69538f6
                  └─490a2f139178
                    └─8e883e7bf499
                      └─1516dd57ebed
                        └─0226fd7d17ee
                          └─da10700e9ac9 Tags: repeater:1.0
                            └─fb52014e02a5 Tags: dependance:1.0
```


Vuelva al directorio `repeater`.

Modifique cualquier cosa en el archivo `Dockerfile`, por ejemplo cambie el intervalo por defecto.


En este momento es cuando debemos compilar la imagen en un número de versión diferente, utilizando por ejemplo el tag `repeater:1.1`. Pero vamos voluntariamente a vulnerar las reglas de buena gestión de las versiones, creando una versión `1.0` diferente de la ya publicada. Conscientes de que la sencilla compilación con el mismo número conduciría - como anteriormente - a identificadores sin etiqueta en la lista, pasamos en eliminar la imagen `1.0`:

Escriba el comando `docker rmi repeater:1.0`.

Compruebe que la imagen no está eliminada, sino que solo se elimina la etiqueta:

 Images/03RI113N.png

¿Qué es lo que sucede? Simplemente, la imagen `dependance:1.0` estaba abajo en la lista y por lo tanto, el comando `rmi` puede provocar una eliminación de la imagen designada por la etiqueta `repeater:1.0`, porque otra imagen la usa. Docker hace bien las cosas y se conforma con eliminar la etiqueta.

 Observe que si se ha hecho cualquier otra cosa, volviendo a compilar directamente una nueva imagen etiquetada 1.0, la arborescencia hubiera mostrado la imagen `dependance:1.0` conectada en la parte de la arborescencia sin etiqueta. La versión 1.0 de `repeater` se pasaría a otra imagen, pero `dependance:1.0` siempre estaría conectado en su imagen de origen. De nuevo una vez más y como sucede con el ejemplo anterior sobre la restauración de un archivo, Docker hace lo necesario para que las imágenes sean perfectamente inmutables y continúen en todos los casos funcionando de la misma manera.

La arborescencia muestra que la imagen `da10700e9ac9` siempre está ahí, incluso si el tag ha desaparecido, lo que permite a la imagen `fb52014e02a5` no quedarse huérfana:

```
└─<missing>
  └─<missing>
    └─<missing>
      └─<missing>
        └─<missing>
          └─2d696327ab2e Tags: ubuntu:latest
            └─f7d11a50b838
              └─df1153da0e1c
                └─dfd6d69538f6
                  └─490a2f139178
                    └─8e883e7bf499
                      └─1516dd57ebed
                        └─0226fd7d17ee
                          └─da10700e9ac9
                            └─fb52014e02a5 Tags: dependance:1.0
```

Ahora vamos a recompilar la imagen `repeater` modificada, asignándole la versión 1.0 (de nuevo, esta no es una buena manera de gestionar las versiones; hacemos esto para mostrar el comportamiento de la cache en caso de modificación de la

imagen básica).

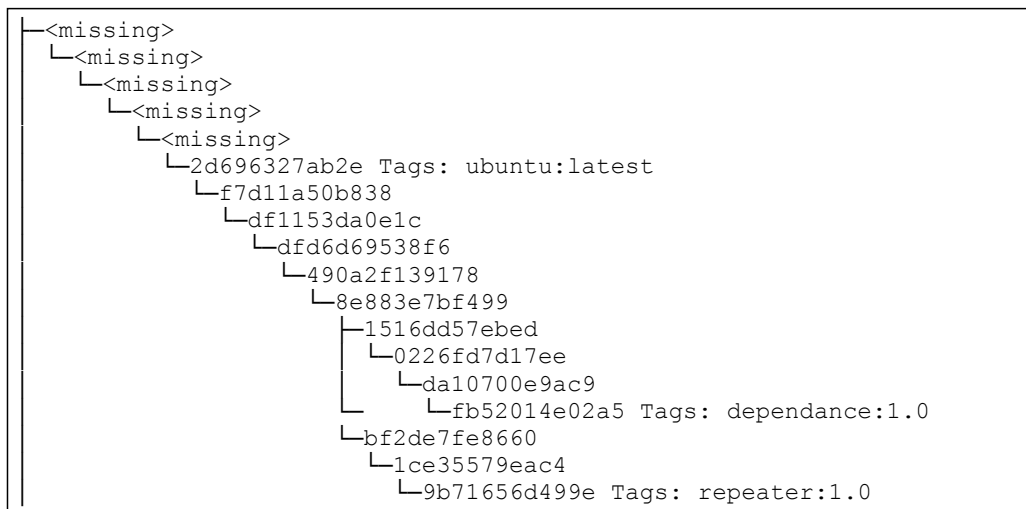
Verifique que continúa estando en el directorio `repeater`.

Lance el comando de compilación `docker build -t repeater:1.0 .` (no olvide indicar el directorio local).

Compruebe que la caché se ha utilizado hasta el comando que haya modificado.

Lance el comando mostrado más atrás para obtener la arborescencia de las imágenes.

Lógicamente, obtenemos una bifurcación con una primera rama que ahora termina por `repeater:1.0`, mientras que la segunda rama termina por `dependance:1.0`, para el momento siempre basada en la imagen básica a partir de la que se ha compilado, y que ya solo queda identificada por su identificador `da10700e9ac9`:



Vamos ahora a volver a compilar la imagen `dependance`, esta vez de manera limpia, es decir, evolucionando su número de versión en consecuencia.

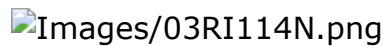
Posiciónese en el directorio `dependance`.

Es inútil modificar el archivo `Dockerfile`. Como la imagen básica se ha modificado, de todas formas habrá una recompilación.

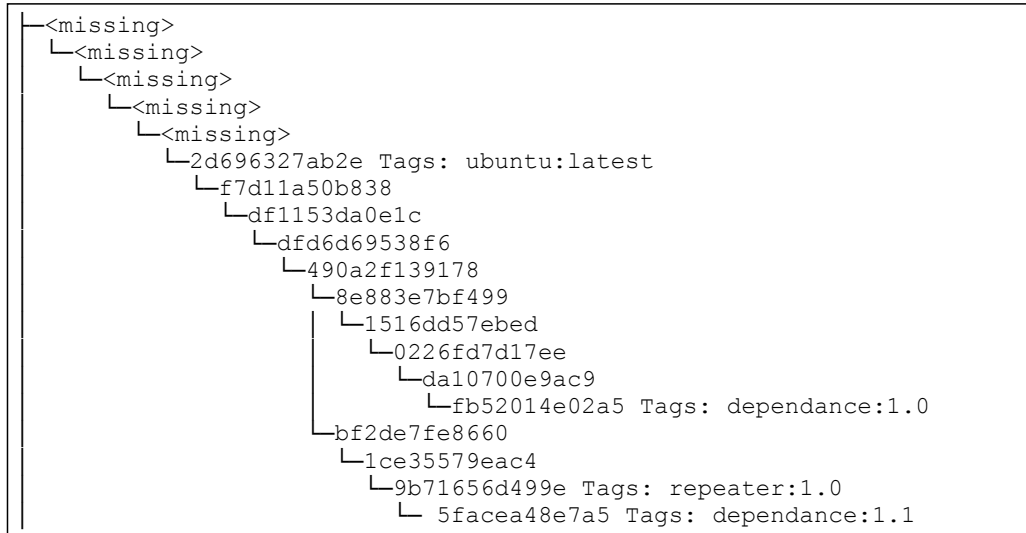
Compile la imagen en un número de versión `1.1`.

La salida muestra que la compilación se hace sin caché, y utilizando la imagen actualizada de `repeater:1.0` (el

identificador utilizado es 9b71656d499e):



Y la arborescencia es conforme a lo esperado, con dependance:1.1 dependiendo de repeater:1.0 y dependance:1.0, dependiendo de la imagen da10700e9ac9 que ya no tiene etiqueta.



Además de mostrar que la caché era inválida durante una modificación de la imagen apuntada por el FROM (incluso cuando esto nunca debería suceder si la gestión de las versiones es correcta), la virtud de este ejemplo es mostrar las funcionalidades implantadas por Docker para que las imágenes sigan siendo inmutables. De nuevo una vez más, esto es una característica fundamental de Docker: la garantía de que cualquier contenedor acabado de desarrollar, funcionará de la misma manera en producción, en un servidor local, en un entorno de tipo cloud, etc.

## f. Invalidación de la caché, modificando el contexto

Los dos casos de invalidación de la caché que hemos visto por el momento son bastante lógicos: si la imagen básica evoluciona, es necesario volver a compilar; si cambia un comando del Dockerfile, es necesario volver a compilar. Pero existe un caso menos evidente y que sin embargo, es importante: se trata de la invalidación de la caché por modificación del contexto.

Más atrás en el presente capítulo, hemos explicado la noción de contexto, remarcando el hecho de que todo el directorio actual estaba disponible para la operación de compilación. De esta manera, en nuestro ejemplo el archivo `heartbeat.sh` está disponible para la operación `COPY`. ¿Qué debería pasar si el archivo `heartbeat.sh` se modifica? Si nos atenemos estrictamente a la aplicación de la invalidación de la caché tal y como la conocemos hasta ahora, ni la imagen básica ni el `Dockerfile` han cambiado, por lo que el impacto sobre la imagen resultante será nulo. Pero la lógica haría que este archivo fundamental para el comportamiento de nuestra imagen hubiera cambiado y la recompilación lo tendría en cuenta.

Esto es lo que ha sucedido y vamos a demostrarlo con las siguientes operaciones:

Elimine todas las imágenes que ha utilizado anteriormente. Si no ha seguido los ejemplos hasta ahora, es suficiente con el comando `docker rmi dependance:1.0 dependance:1.1 repeater:1.0`.

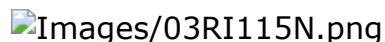
Sitúese en el directorio `repeater`.

Compile la imagen una primera vez con el tag `repeater:1.0`.

Modifique simplemente el archivo `heartbeat.sh`, añadiendo por ejemplo una línea en mitad del archivo.


Compile de nuevo la imagen, esta vez con el tag `repeater:1.1`.

La salida debe ser parecida a la siguiente, en la que se muestran las primeras etapas, en las que se ha utilizado la caché, pero la etapa nº5 que se corresponde con el comando `COPY`, no la ha podido utilizar:

Images/03RI115N.png

Docker ha detectado que el contexto de compilación (el contenido del directorio actual, en este caso de nuestro archivo `heartbeat.sh` está en la misma ubicación que el archivo `Dockerfile`) ha cambiado y por lo tanto ha invalidado a la caché con el comando `COPY`. Además, sucedería lo mismo con el comando `ADD`, muy parecido a `COPY` en su funcionamiento.

Observe que no ha sido necesario cambiar el nombre del archivo: es la detección del cambio de su contenido, lo que ha provocado la recompilación de una parte de las imágenes intermedias.

 Es la emisión de este contexto, apuntado al inicio de los mensajes mostrados durante la compilación (línea `Sending build context to Docker daemon`).

En la actualidad, los tres casos mencionados son los únicos que conducen a una invalidación de la caché. En la sección siguiente, vamos a ver en qué parte se necesita prestar atención para determinados usos.

### 3. Consecuencias de la escritura de Dockerfile

#### a. El problema en las operaciones no idempotentes

El problema se presenta en los comandos Docker que no son idempotentes. Un comando idempotente es una operación que conduce sistemáticamente al mismo resultado cuando se lanza en el mismo contexto. Por ejemplo, la asignación de una variable de entorno por el `Dockerfile` es idempotente porque producirá el mismo resultado cada vez, a saber, crear una variable de entorno con un valor dado en el contenedor que utiliza la imagen generada. A la inversa, el comando `date` es un ejemplo de operación no idempotente, porque en función del momento en el que se llame, el resultado será diferente.

¿Cuál es la consecuencia para nuestro `Dockerfile` y su compilación? Para entender el problema, tomemos el ejemplo de comando no idempotente al pie de la letra y creemos una imagen dependiente del comando `date`.

Cree un directorio `testcache` en cualquier sitio, salvo dentro de un directorio que contiene un archivo `Dockerfile`.


Sitúese en este directorio.

Cree un archivo `Dockerfile` que contendrá el siguiente contenido:


```
FROM ubuntu:latest
RUN date +%N > /tmp/moment
ENTRYPOINT ["more"]
CMD ["/tmp/moment"]
```

La imagen solo almacena el número de nanosegundos correspondientes al momento presente en un archivo, y va a mostrar el contenido de este durante la instanciación de un contenedor.

La compilación de la imagen debería dar un resultado como el siguiente:

images/03RI116N.png

Y si ejecutamos un primer contenedor, el resultado devolvería un número aleatorio de nueve cifras:


images/03RI117N.png

Observemos ahora lo que provoca una recompilación del `Dockerfile`.


Elimine el contenedor lanzado anteriormente.

Lance de nuevo la compilación del `Dockerfile`.

La compilación debería mostrar que las tres etapas utilizan la caché:

images/03RI118N.png

Y lógicamente, si lanzamos un segundo contenedor, veremos el mismo resultado para el resultado del comando `date`, aunque el identificador del contenedor muestre que estamos en otra instancia:

images/03RI119N.png

Al contrario de lo que sucede con los comandos `COPY` y `ADD`, que pueden provocar la invalidación de la caché si el contexto cambia, el comando `RUN` no modificado se recupera sistemáticamente desde la caché, aunque su ejecución pueda provocar un comportamiento diferente de la imagen.

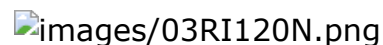
Si extrapolamos este comportamiento a los comandos más industriales y realizamos una llamada de `apt-get update`, por ejemplo, en lugar de nuestro comando artificial de almacenamiento de una fecha, las consecuencias son más embarazosas. En efecto, el objetivo de `apt-get update` es

actualizar los almacenes en los que Ubuntu va a buscar las actualizaciones de las aplicaciones o del sistema. La consecuencia del cacheado de este comando es que, durante todas las compilaciones después de la primera, el comando ya no se ejecutará más, pero su resultado se cacheará en la caché de las imágenes. Las actualizaciones más recientes de los paquetes Ubuntu se ignoran. Imaginamos las consecuencias desastrosas de no tener en cuenta un patch urgente de seguridad...

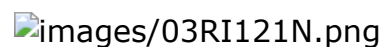
De la misma manera, durante la primera compilación de un `Dockerfile` que lanza un comando de tipo `git clone [dirección del almacén]`, recupera el contenido del almacén Git asociado, pero sigue estando bloqueado en esta versión del código, mientras que el comportamiento deseado es que cada compilación tome la versión del código actualizada. También es habitual tener que descargar los scripts o el contenido en un `Dockerfile`, normalmente utilizando el comando `curl`. Estas operaciones presentan los mismos problemas.

## **b. Solución del problema de la caché**

El enfoque más directo para ajustar estos problemas, consiste en compilar la imagen con la opción `--no-cache`, lo que fuerza la invalidación de todos los niveles de caché.



Una prueba rápida permite comprobar que el comportamiento ha cambiado, esta vez:



El enfoque de desactivación completa de la caché, no es particularmente elegante y sobre todo, hace necesario (salvo si la operación se automatiza) que la persona que compila piense en utilizar la opción. También es fácil pasar por alto información si la caché se ha utilizado o simplemente, no volver a recordar que la caché tiene un problema para esta imagen en particular. Para terminar, el tiempo de compilación de la imagen nunca se reduce lo que por ejemplo, puede penalizar en un proceso muy iterativo de desarrollo.

En algunos casos, el uso de números de versión para los contenidos por ejemplo nos puede ayudar, si se ejecuta un



comando `git checkout` después del comando `git clone`. Tomemos el ejemplo de un archivo `Dockerfile` como sigue:


```
FROM ubuntu:trusty
RUN git clone [dirección del almacén]
WORKDIR [directorio recuperado]
RUN git checkout v1.0.0
RUN [operación de compilación]
```

Cambiar la versión de la cuarta línea va a invalidar la caché. El problema es que todas las operaciones que vienen después se realizarán de nuevo, pero no las anteriores. La recuperación de las fuentes se hace durante la operación `git clone`. Esto va a llevar a la implantación de la caché y la imagen intermedia se generará durante la primera compilación, con el código antiguo. Incluso hay opciones de que el comando `git checkout` sobre una nueva versión, no funcione porque este todavía no existe.

El truco, recomendado en las buenas prácticas de Docker ([https://docs.docker.com/articles/dockerfile\\_best-practices/](https://docs.docker.com/articles/dockerfile_best-practices/)), consiste en agrupar los dos comandos en una única línea:

```
FROM ubuntu:trusty
RUN git clone [dirección del almacén] \
    && cd [directorio recuperado] \
    && git checkout v1.0.0
RUN [operación de compilación]
```

De esta manera, durante la compilación de otra versión Docker verá que el comando de `Dockerfile` ha cambiado globalmente (recuerde que un sencillo espacio es suficiente para invalidar el código), y ejecutará de nuevo la operación de clonado del almacén Git, recuperando de esta manera todo el código fuente actualizado.

 La razón por la que se utiliza el comando `cd` para situarse en el directorio creado por el comando `git clone`, es que la operación `WORKDIR` insertada en medio impide fusionar las dos operaciones `RUN`. También es posible crear el directorio con antelación y hacer que apunte a `git clone`, para evitar de esta manera realizar una operación de `RUN` en el comando `cd`, lo que no es muy legible.

Este enfoque basado en la modificación de un número de versión, es más limpio que la opción vista más atrás (`--no-cache`). En efecto, permite una desactivación selectiva de la caché


únicamente para los comandos afectados - así como los siguientes. Pero además, este método es más limpio desde el punto de vista del control de la necesidad de volver a compilar, porque el administrador de la imagen que interviene para actualizar la versión, es perfectamente consciente del ciclo de vida de su imagen.

Nos encontramos con el mecanismo en determinados feedbacks sobre compilaciones automáticas de imágenes, donde los archivos `Dockerfile` algunas veces hacen referencia a un archivo cuyo nombre se genera y cambia cada vez que es necesario. También podemos utilizar el hecho de que una modificación del contexto invalida la cache, recuperando un script en el contexto en cuestión. Llevando este razonamiento hasta el final, también se pueda considerar transferir todo el código fuente a compilar por el contexto, en lugar de ir a buscarlo de manera remota con un `git clone`. Por lo tanto, el `Dockerfile` está en el directorio que contiene todo el código fuente a utilizar. En este caso será necesario limitar el volumen por razones de rendimiento. En efecto, todos los archivos se vuelven a copiar en el contexto durante la compilación, lo que pueda ralentizar mucho la operación en caso de volúmenes importantes.

Para terminar, el enfoque siempre es el mismo cuando la caché se deba invalidar, es decir, llamar a los comandos de tipo `apt-get`: lo más sencillo es pasar la versión exacta del paquete a instalar, de tal manera que la invalidación de la caché sea natural cuando el administrador de la imagen actualiza esta dependencia. Normalmente vemos en los `Dockerfile` encadenamientos como el siguiente:

```
RUN apt-get update && apt-get install -y \  
    automake \  
    curl=7.35.0* \  
    git \  
    mercurial
```

La separación línea a línea con los símbolos "`\`" para saltar de línea y en orden alfabético de los paquetes, son recomendaciones dadas por Docker. Esta escritura hace más sencillo añadir paquetes, sin correr el riesgo de duplicaciones.

 Cuando se usa `apt-get install`, especificar el número de versión es una buena costumbre. Después de todo, es lógico conocer este número cuando se compila una nueva imagen. En efecto, hemos visto más arriba que era una buena práctica ofrecer un tag para cada compilación. ¿Cómo saber si

necesitamos una nueva compilación si no sabemos cuáles son las dependencias que han evolucionado? Cuando las dependencias son muy numerosas, la práctica cada vez más preferida es la de separar los ciclos de vida y compilar después de cada modificación en el código, teniendo en cuenta sistemáticamente las últimas dependencias y verificando que no plantean un problema, usando una batería de pruebas de aceptación. Pero también existen casos en los que la recompilación se debe hacer para estar actualizado respecto a una dependencia que ha recibido un patch o una evolución principal. El número es perfectamente conocido y el ciclo de vida de la dependencia controlada.


### **c. Efectos beneficiosos del número y tamaño de las imágenes**

La agrupación de comandos mostrada anteriormente, presenta beneficios adicionales. Tomemos como ejemplo el siguiente extracto imaginario de `Dockerfile`:

```
WORKDIR /tmp
RUN curl http://www.gouigoux.com/big-archive.tgz -o big-archive.tgz
RUN tar zxvf big-archive.tgz
RUN rm big-archive.tgz
WORKDIR /tmp/big-archive
RUN make
WORKDIR /
RUN rm -fr /tmp/big-archive
```

Es sensato eliminar los archivos temporales, pero como cada comando se realiza en una etapa separada y cada uno resulta en una imagen correspondiente de su estado, la imagen correspondiente a la etapa nº2 en nuestro ejemplo, necesariamente va a contener archivos extraídos del archivo. Y la eliminación en la etapa nº3 no reducirá en nada el tamaño de esta imagen; simplemente creará otra imagen (realmente pequeña), que almacenará la información que los archivos son eliminados a continuación.

El tamaño total de las imágenes, sigue la siguiente composición:

images/03RI68.png

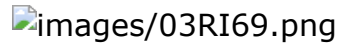
Por esta razón, es más limpio escribir el `Dockerfile` como sigue:

Al final de la ejecución de este comando, se ha generado una

```
WORKDIR /tmp
RUN curl http://www.gouigoux.com/big-archive.tgz -o big-archive.tgz \
    && tar xzvf big-archive.tgz \
    && rm big-archive.tgz \
    && cd /tmp/big-archive \
    && make \
    && cd .. \
    && rm -fr /tmp/big-archive
```

única imagen en lugar de siete y sobre todo, solo contendrá los archivos generados por la compilación (`make`) y ninguno de los archivos temporales.


La composición del comando solo genera una imagen, lo que se podría representar como sigue:



El mismo efecto de apilamiento de las imágenes, aparece cuando se utiliza el comando `ADD` para recuperar el contenido de una URL, que es necesario eliminar posteriormente. Por lo tanto, se recomienda sustituir este enfoque por una operación `RUN` que empiece por efectuar una descarga con `curl` o `wget`, como acabamos de hacer.

Además es posible ir más allá, prescindiendo de archivos que son temporales. Para esto, podemos utilizar los pipes Linux para enviar el contenido a la herramienta `tar`. La opción `-o` es inútil, así como el nombre del archivo en el comando `tar` y el comando de eliminación del archivo, que no ha existido en ningún momento:

```
WORKDIR /tmp
RUN curl http://www.gouigoux.com/big-archive.tgz | tar xzvf \
    && cd /tmp/big-archive \
    && make \
    && cd .. \
    && rm -fr /tmp/big-archive
```

 De manera general, se recomienda durante las primeras compilaciones vigilar el tamaño de las imágenes. Con la práctica, las pérdidas de recursos saltan a los ojos. Si a pesar de todo, se produce un error en su proceso, sigue siendo posible eliminar una pila de imágenes en una sola, utilizando el truco ofrecido en la sección "Flatten Your Image", del excelente artículo de Century Link Labs sobre la optimización de las imágenes Docker (ver todos los detalles en <https://www.ctl.io/developers/blog/post/optimizing-docker-images/>).

#### **d. Programación de los comandos en el Dockerfile**

Para terminar, algunas buenas prácticas de programación también pueden ayudar a conservar las imágenes compactas.

En primer lugar, para beneficiarse el mayor tiempo posible de la caché, es interesante poner lo más atrás posible en el archivo `Dockerfile` los comandos que no cambian, como por ejemplo `ENTRYPOINT` o `CMD`. Esto pueda parecer poco intuitivo, ya que tal y como se describe el proceso que se va a lanzar después de todas sus fases de preparación, la tendencia natural es ponerlas al final del archivo. Pero esto no es obligatorio en Docker.

Otra mejora sobre las imágenes que necesitan la compilación de código fuente, es poner antes las operaciones de restauración de las dependencias y después solo volver a copiar el código fuente y compilarlo. Este modo de funcionamiento es particularmente útil con los frameworks que recuperan las dependencias en los almacenes de Internet (Node.js, .NET Core con las dependencias NuGet, etc.), porque una ruptura de la caché en este caso, puede provocar pesadas descargas. Además, si la empresa no ha implementado una caché de módulos, estas descargas se multiplicarán por el número de desarrolladores, lo que puede plantear problemas de rendimiento, así como de consumo de ancho de banda.

De manera general, la velocidad de compilación de una imagen viene de la unión de muchos trucos y buenas prácticas. Por lo tanto, aprender constantemente es necesario, lo que también permite estar al corriente de los nuevos enfoques de reducción masiva de las imágenes por compilación nativa y producción de imágenes reducidas a un único ejecutable, sin las herramientas que han permitido crearla. La página <http://container-solutions.com/lean-go-containers-multi-stage-dockerfiles/>

describe este mecanismo para el lenguaje Go, utilizando las compilaciones Docker multiestado, obteniendo como resultado una imagen funcional que pesa solo 1,55 MB. La reciente puesta en disposición de un compilador nativo y de un linker para .NET Core, también hace posible este tipo de implantación ultraligera para esta otra plataforma de desarrollo moderna.

## 4. Consecuencias de la elección de las imágenes básicas

### a. La imagen básica correcta

Las consecuencias del mecanismo de caché y de manera general, del modo de funcionamiento de las imágenes que se apilan las unas sobre las otras, son de varios tipos.

En primer lugar, es fundamental no multiplicar imágenes básicas. Salvo si hay una necesidad particular, se recomienda elegir una distribución y atenerse a ella lo máximo posible. Si todas las imágenes se construyen a partir de una imagen `debian:wheezy`, los 100 MB correspondientes se descargan una vez para todas y solo la parte superior de las imágenes fluctuará. En una `ubuntu:xenial`, hablamos alrededor de 125 MB, pero incluso si la imagen es más grande, lo importante sigue siendo sobre todo no multiplicar las distribuciones y las versiones. Si alguna de sus imágenes están basadas en una `ubuntu:xenial`, otras en una `ubuntu:trusty`, otras en `debian:wheezy` y las últimas en una `centos:latest`, está multiplicando la carga en el disco (lo que puede resultar molesto), así como sobre el ancho de banda (lo que generalmente es muy molesto, porque está relacionado directamente con el rendimiento).


La elección de la versión de la imagen, también es importante. Utilizar el defecto `latest` (la última), conduce al cabo de algún tiempo a la presencia de varias versiones efectivas en su caché de imágenes locales, si el tag `latest` se mueve mucho. Aquí de nuevo es más limpio validar una versión dada y ajustarse a ella. Salvo necesidad particular, cuando utilice una imagen `ubuntu`, es lógico utilizar la última LTS y el equipo que mantiene la imagen oficial, hace justamente el esfuerzo de no mover la etiqueta `latest` dentro de las versiones menores de la LTS, de manera que continúen los patches, pero sin imponer un cambio más importante de imagen a los usuarios.

Para terminar, no es necesario utilizar imágenes tan grandes en todos los casos. Las imágenes `ubuntu` y `debian` son muy conocidas, porque ofrecen distribuciones casi completas, aunque se haya hecho un esfuerzo enorme de reducción de tamaño, respecto a las versiones propuestas durante la instalación como OS principal de una máquina física y que ocupan un DVD de 4,7 GB. Pero en el 20% de los casos más sencillos, que representan el 80% de los usos, una imagen reducida como `busybox` puede servir perfectamente y solo pesa algunos MB.

Los editores de las distribuciones, prestan atención al tamaño de sus imágenes y el tamaño de la imagen básica Ubuntu ha bajado en los últimos años. Incluso si 125 MB parece todavía desmesurado para los usos de micro-servicios, conviene recordar que durante la primera versión de este libro (2015), la misma imagen pesaba 190 MB. La reducción también es visible en Debian. La imagen ya era más ligera con 125 MB y ahora no ocupa más de 100.

Debian va incluso más allá con sus imágenes de tipo slim, es decir, ligeras o delgadas según la traducción inglesa adoptada. Una Debian Jessie en versión slim solo ocupa 79 MB y la más reciente, Debian Wheezy en versión slim, baja hasta 47 MB. Todavía no estamos en tamaños ultra-reducidos de únicamente 4 MB, como una imagen Alpine, pero el esfuerzo ya es enorme.

Se debe reproducir el mismo esfuerzo de concisión en las imágenes básicas para los contenedores Windows, una imagen `microsoft/windowsservercore` que pesa varios GB, mientras que una imagen `microsoft/nanoserver`, que es hoy en día la alternativa más restringida posible para los contenedores Windows, todavía ocupa varios centenares de MB.

 No se dará ninguna cifra concreta porque esta operación de adelgazamiento todavía continúa y los tamaños varían mucho en el momento de escribir este libro. Se anima al lector a experimentar sus conocimientos para encontrar el tamaño virtual de estas imágenes.

## **b. Su propia imagen básica**

Otro medio eficaz de ganar espacio, es establecer una capa de imagen para todas las funcionalidades que utilice normalmente. Por ejemplo, un equipo de desarrollo va a estar acostumbrado a tener determinadas herramientas en las máquinas de ejecución

de sus aplicaciones. Un equipo orientado a la web, necesitará Nginx NodeJS, AngularJS, etc. Si las imágenes se construyen normalmente para probar las aplicaciones desarrolladas por este equipo, una buena práctica es construir una imagen común de "desarrollo web", que servirá de base para todas estas máquinas de prueba que normalmente tienen una duración de vida muy limitada (el tiempo de las pruebas, porque otra versión llega pocas horas más tarde si el equipo trabaja de manera just-in-time).

El ahorro en términos de rendimiento es importante para trabajar en la integración continua. Pocos segundos para descargar, instalar y configurar las dependencias, representan un coste al final del año cuando decenas de imágenes de prueba se han generado cada día. La comodidad de los equipos también mejora.

Por ejemplo, imaginemos una imagen para establecer un servidor web escrito en Python, que utiliza un equipo de desarrollo para lanzar varias veces al día las pruebas en cada uno de los puestos de desarrollo. Estas pruebas normalmente se ejecutarían en un contenedor arrancado únicamente para validar que las modificaciones del código de cada desarrollador no presenten ningún problema.

Un Dockerfile como este, sería como sigue:

```
FROM ubuntu:trusty

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask

COPY webtest.py webtest.py

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

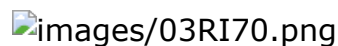
Aprovechamos la presentación de este Dockerfile para recordar las buenas prácticas:

- Utilización de una imagen básica con una versión explícita.
- Agrupación de los comandos de instalación en una única operación RUN.
- Disposición en las líneas para facilitar su lectura.



- Añadir un espacio antes del símbolo `\` de retorno de línea, también para esclarecer el texto durante la lectura.
- Respeto el orden alfabético para hacer visibles las eventuales repeticiones durante futuras adiciones.
- Utilización preferente de la operación `COPY` en lugar de `ADD`.
- Posicionamiento del comando de copia del archivo origen después del comando que gestiona las instalaciones de las herramientas, mucho más pesadas, de manera que no se invalide este nivel de caché cuando se realice una modificación del código fuente (es un aspecto esencial).
- Separación `ENTRYPOINT` y `CMD`.

Un vistazo a la estructura de la imagen utilizando el comando `history`, muestra que las herramientas instaladas ocupan más de 160 MB:



Claramente, la utilización de la caché es obligatoria. La idea de compartir una imagen básica que contiene estas herramientas entre todas las personas de un equipo de desarrollo, pueda evitar pérdidas de tiempo en las reconstrucciones. De esta manera, todos los desarrolladores pueden utilizar una imagen básica actualizada, sin tener que pasar por las etapas de compilación. El ahorro no es excepcionalmente elevado en este caso, porque cada máquina de desarrollo tiene su propia caché de imágenes de todas maneras, pero veremos en un siguiente capítulo un segundo ejemplo con un caso particular de compilación de aplicación, que hace el montaje con una imagen básica "desarrollo", particularmente útil.

Sin anticipar demasiado sobre este ejemplo, se trata de un entorno .NET donde NuGet recupera las dependencias y se descargan de manera dinámica en función del código fuente. Como la carga del código fuente en la imagen se hace antes de la recuperación de las dependencias por NuGet, esta última etapa de la construcción siempre empieza desde cero después de cada modificación de código, mientras que en una gran mayoría de los casos, se trata de las mismas librerías. El hecho de preparar una imagen básica con el código "estándar" (resultante de la descarga de todas las librerías más comunes), permite ganar mucho tiempo en el resto de compilaciones de imágenes posteriores.

Volveremos más en detalle sobre esta buena práctica durante la implementación de la aplicación de ejemplo, que nos va a servir para demostrar la gestión de un despliegue Docker más industrial, en un capítulo posterior.

## 5. Arborescencia recomendada

### a. Ventajas de una arborescencia tipo

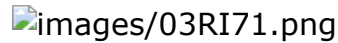
Cuando se compila una imagen a partir de un `Dockerfile`, el directorio destino se copia completamente durante la compilación, porque se corresponde con lo que hemos llamado el contexto. Es en este contenido donde las funciones `ADD` y `COPY` van a buscar los archivos necesarios para la creación de las imágenes y estas no tienen la posibilidad de ir a buscar otros archivos a un directorio arbitrario del sistema de archivos.

Esto induce un modo de almacenamiento de los archivos necesarios para la creación de una imagen que es bastante estándar, a saber, un directorio que contiene todo lo que será necesario para la compilación, con `Dockerfile` en el primer nivel de este directorio. También es habitual establecer un archivo de información, así como un archivo de licencia si es necesario, en el mismo lugar.

El enlace con el desarrollo de software es muy fuerte, porque Docker normalmente se utiliza para el despliegue de aplicaciones después de que se compilen. En un capítulo posterior, volveremos sobre todas las contribuciones de Docker a ALM (*Application Lifecycle Management*), es decir, la gestión del ciclo de vida de un programa informático, desde su compilación a su mantenimiento) y, en particular, a la integración continua, pero esta sección sobre la estructura arborescente de los proyectos Docker no se puede abordar sin explicar por qué esta estructura está íntimamente ligada a la del código fuente.

Si usamos como ejemplo un proyecto de GitHub, aquí también existe una estructura estándar, con un archivo `README.md` (formato Markdown) en la raíz, así como el archivo `.gitignore` y el eventual archivo `LICENSE.md`. Una buena práctica es fusionar el directorio que contiene el `Dockerfile` con el que contiene el proyecto que va a servir para ofrecer los entregables que se integrarán en la imagen Docker durante la compilación.

De manera reducida, esta arborescencia "tipo" es la siguiente:



Las ventajas de este almacenamiento en un único directorio, son numerosas. En primer lugar, la compilación es completamente autónoma y `Dockerfile` no necesita ninguna otra referencia que la que se encuentra en el directorio actual, lo que facilita su desplazamiento. Además, como la gestión de código fuente trabaja generalmente en proyectos contenidos también en un único directorio, es relativamente fácil hacer que ALM realice la compilación del código y después la generación de la imagen en un mismo script, que de esta manera facilita la gestión de versiones. Para terminar, esto hace que sea sencillo incluir el `Dockerfile` directamente en el almacén GitHub y, de esta manera, gestionarlo exactamente como si se tratara de un archivo de código fuente.

## **b. Integración de los archivos**

Este enfoque se relaciona con la noción de Infrastructure As Code, donde la gestión de la infraestructura de despliegue de una aplicación de software se escribe en un lenguaje cercano a un lenguaje de desarrollo. Esto permite mejorar la calidad del despliegue, porque la operación está totalmente automatizada y no depende de etapas manuales. El riesgo de errores se reduce y sobre todo, el tiempo completo de instalación de un entorno se reduce mucho, lo que permite probar de manera más normal.

La mezcla entre los dos dominios también se relaciona con la noción de DevOps, donde los desarrolladores y el personal operativo están llamados a trabajar juntos de manera mucho más estrecha. Tradicionalmente, los desarrolladores terminan su trabajo, y proporcionan los entregables al personal operativo, que lo deben instalar y explotar. En un enfoque DevOps, donde el desarrollo se hace de manera más continua, la instalación y la explotación están previstas desde el inicio y completamente como etapas del proceso de desarrollo. Por lo tanto, hay una continuidad total entre la creación de código fuente, su compilación, su validación por las pruebas y su despliegue en el entorno destino. El objetivo de las herramientas de integración continua es hacer que la modificación de un archivo de código fuente se repercuta de manera automática al entorno final.

Esta manera de hacer, también provoca las buenas prácticas en la escritura de los archivos `Dockerfile`. Por ejemplo, se recomienda basarse en la estructura de directorios para los diferentes comandos de generación de la imagen, porque

constituye una convención compartida. Imaginemos el siguiente archivo Dockerfile:

```
FROM ubuntu:trusty

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask

COPY src/webtest.py webtest.py
COPY src/mathapi.py mathapi.py
COPY src/math.py math.py

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

Si un desarrollador añade un archivo de código fuente en la solución, también será necesario modificar el archivo Dockerfile. Si se olvida la modificación, puede ser que la aplicación funcione correctamente en su puesto, pero es posible que la funcionalidad correspondiente no tenga la imagen Docker generada. El personal operativo deberá intervenir para encontrar por qué el entorno no funciona como se espera. Realizado su diagnóstico, se deberá añadir el nuevo archivo de código fuente en Dockerfile o pedir al desarrollador que solucione su olvido.

La solución consiste en un Dockerfile en que nos basamos, por convención, en el conjunto del contenido del directorio `src`. Añadir un archivo fuente se tiene en cuenta automáticamente, lo que hace que el funcionamiento de DevOps sea más fluido. El Dockerfile al final es más sencillo:

```
FROM ubuntu:trusty

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask

COPY src/ .

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

### **c. Limitación del contexto**

Preste atención con no caer en el extremo opuesto, de tener en cuenta de manera indiferente todo el contenido. Por defecto, todo el contexto se copia durante la compilación de la imagen, lo que hace necesario vigilar el volumen del directorio que contiene el `Dockerfile` y todo lo necesario.

Si este directorio se fusiona con el de un proyecto de software, como habíamos recomendado más atrás, no es raro que la compilación del código fuente (no de la imagen) resulte en la creación de numerosos archivos temporales y logs, inútiles para el funcionamiento de la imagen. Por lo tanto, es necesario limpiar lo que pase en el contexto.

El método más elegante para hacer esto es establecer un archivo `.dockerignore` en el que se especifiquen todos los patterns correspondientes a los archivos a no tener en cuenta durante la compilación o de manera más precisa, durante el establecimiento del contexto.

Un contenido típico sería el siguiente:

```
.dockerignore
.git
.gitignore
bin
logs
```

La sintaxis completa para la exclusión de los archivos y directorios es el de la función `Match` del lenguaje Go, con el que se escribe Docker. La referencia es <http://golang.org/pkg/path/filepath/#Match>. Observe que es una buena práctica listar el archivo `.dockerignore` en sí mismo: parece difícil imaginar un caso donde podría servir en la imagen compilada porque contiene justamente información útil durante la compilación de esta imagen.

## 6. La pregunta del proceso único

### a. Principio general

La filosofía general de los contenedores, en particular en las arquitecturas de micro-servicios, es que un contenedor debe contener un único servicio de aplicaciones.

La aplicación más rigurosa de este principio lleva a autorizar un único proceso en un contenedor (por tanto, con el PID 1). La ventaja de este enfoque es la sencillez de la gestión, porque la


gestión de los contenedores se alinea perfectamente con las aplicaciones.

## **b. Excepción al principio general con Supervisor**

Docker no evita que se lancen múltiples procesos en el mismo contenedor, lo que es lógico porque lo contrario impediría hacer funcionar cualquier aplicación que utilice el comando `fork`, que genera un proceso hijo resultado del proceso actual. Algunos han utilizado este enfoque para establecer imágenes "completas" que contienen, por ejemplo, un CMS (*Content Management Software*: software de gestión de contenidos) así como su base de datos de persistencia de artículos y otras entidades publicadas.

Supervisor es un proceso de gestión de ejecución de otros procesos, que permite establecer fácilmente este tipo de cosas. Imaginemos, por ejemplo, que queremos establecer un servidor web Apache, así como una base de datos PostgreSQL en la misma imagen. Para esto, además de los comandos de instalación tal y como hubiéramos escrito si hubiéramos creado dos `Dockerfile` separados, añadimos la siguiente operación:

```
RUN apt-get update && apt-get -y install supervisor
```

 Piense en fusionar esta instalación con el resto de paquetes a instalar, para no crear capas de imágenes inútiles.

La siguiente etapa consiste en crear un archivo `supervisord.conf`, con un contenido parecido al siguiente:

```
[supervisord]
nodaemon=true

[program:pgsql]
command=/usr/local/pgsql/bin/pg_ctl start -l logfile -D
/usr/local/pgsql/data
user=postgres
autorestart=true

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec
/usr/sbin/apache2 -DFOREGROUND"
```

La primera opción permite no arrancar Supervisor en modo demonio, sino como proceso en primer plano, lo que es necesario para Docker, el ciclo de vida de un contenedor estaba relacionado al del proceso principal con identificador 1. Después, las secciones permiten definir los procesos a lanzar en la puesta en marcha. En la primera, se da el comando de ejecución de la base de datos PostgreSQL, que se lanzará en el usuario `postgres` y vuelve a arrancar de manera automática en caso de problemas. En el segundo, se configura el servidor Apache.

Para que este archivo se tenga en cuenta en nuestra imagen, será necesario volver a copiar el archivo del contexto en la imagen durante la compilación:

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

Para terminar, la operación `CMD` deberá lanzar Supervisor en lugar de un único proceso:


```
CMD ["/usr/bin/supervisord"]
```

Cuando un contenedor se lanza a partir de la imagen compilada con el `Dockerfile` descrito anteriormente, los dos procesos se arrancarán y se exponen los puertos, si las operaciones correspondientes se han tomado de `Dockerfile` separados.

### c. Crítica

Este enfoque es criticable, en el sentido en que se hace imposible detener el servidor web, sin detener la base de datos. Si además, tal y como sucede con el servidor web, la base de datos está expuesta al exterior a través de su puerto estándar, esto quiere decir que otros procesos podrán escribir datos en PostgreSQL. De un golpe, se hará más complicado arrancar el servidor web recortando la imagen, porque habrá dependencias con otras imágenes. Si al menos la base de datos está limitada al uso interno por la imagen, esto permitirá evitar este problema.

Hay polémica respecto al uso correcto de Supervisor y algunos consideran su uso contrario al espíritu de Docker, fomentando la presencia de un servicio único en una imagen.

 Esta polémica está contaminada por el hecho de que, cuando se utiliza Supervisor en una imagen Docker, termina tomando el proceso de identificador 1 en la imagen. Sobre una máquina física, este proceso se asocia al programa encargado de la

puesta en marcha de todos los servicios, llamados `init`, y existe otra polémica sobre la reciente utilización de `systemd` como nueva herramienta `init` sobre Debian y Ubuntu. En la práctica, los dos aspectos no tienen nada que ver.

Como toda polémica en el mundo Linux, donde los expertos se sienten tentados por crear bifurcaciones para avanzar por su propio camino, esta queda resuelta rápidamente cuando las buenas prácticas se impongan entre los usuarios. El uso siempre genera su propia ley.

Han pasado dos años desde esta comprobación en la primera edición del presente libro y la polémica poco a poco se ha ido extinguiendo. Las arquitecturas de micro-servicios fomentan el comportamiento restringido, en forma de caja negra y evitando abrir más medios de comunicación que los que sean absolutamente necesarios para realizar la tarea, los contenedores se orientan cada vez más a los sistemas mono-proceso.

El caso mencionado anteriormente, sobre la necesidad de un servidor SSH para poder acceder al contenido del contenedor, queda patente por el hecho de que se trata de una mala práctica de seguridad; el comportamiento de un contenedor solo se debe observar desde los puertos y volúmenes que abre y los logs que produce. Para los casos raros en los que se deba lanzar temporalmente un proceso anexo (por ejemplo un programa de copia de seguridad o diagnóstico), el comando `docker exec` es suficiente, y securiza bien el conjunto porque solo se puede lanzar por una persona con acceso, no solo al contenedor, sino también al demonio Docker que soporta.

#### **d. Enfoque intermedio**

Más allá de esta polémica sobre la posibilidad de lanzar puntualmente procesos adicionales, pueda tener sentido en algunos casos concretos tener un proceso primario que represente la funcionalidad principal de la imagen y que se acompañe por un proceso esclavo o herramientas, que permiten al primero funcionar mejor.


En este sentido Docker publica la página de documentación sobre la utilización de Supervisor ([https://docs.docker.com/engine/admin/multi-service\\_container/](https://docs.docker.com/engine/admin/multi-service_container/)), con uno de los dos procesos que es el demonio SSH, y el otro el servidor web Apache. Hay casos particulares, por supuesto no aceptables en producción, pero sin



embargo existentes, que pueden justificar una imagen donde el proceso "principal" sería Apache Web Server, pero que desee dejar que un administrador se conecte al contenedor. El demonio SSH es un proceso secundario. Desde el punto de vista técnico, el proceso principal (de PID 1) es Supervisor. Pero la lógica se respeta con un servicio principal, que es la exposición del contenido web.

Otra marca del reconocimiento oficial de este modo de funcionamiento por Docker, es la aparición del comando `exec`, citado más atrás y que permite arrancar un proceso en un contenedor ya lanzado. Este comando es particularmente útil para supervisar un contenedor de manera introspectiva (por ejemplo, realizar un `tail` en un archivo de log), lo que una vez más, constituye una actividad secundaria y no contraviene el espíritu de servicio único del contenedor. Por supuesto, sería más limpio hacer que el contenedor exportara todos los mensajes necesarios para su supervisión desde el exterior, pero la ejecución de un proceso de control, en fase de control, no tiene nada de escandaloso.

A efectos puramente informativos, a continuación se muestra un ejemplo de uso del comando `docker exec`, para recuperar la copia de seguridad de una base de datos MongoDB:

 Images/03RI122N.png

Para terminar en este punto casi filosófico, Phusion da una buena explicación (en <https://github.com/phusion/baseimage-docker#wait-i-thought-docker-is-about-running-a-single-process-in-a-container>) del hecho de que varios procesos no quieren decir necesariamente que se pierda de vista el aspecto mono-servicio de un contenedor.

## Para ir más lejos

Una última observación en este capítulo sobre la composición de las imágenes: hay numerosas herramientas informáticas que necesitan enfoques similares de descripción del contenido de la aplicación de entidades autónomas. Por ejemplo, herramientas de composición de máquinas virtuales como Vagrant o bien herramientas de descripción de despliegues en las plataformas cloud.

Hay una herramienta, llamada Packer, que permite describir de manera unificada la composición de estas entidades y después, gracias a los plug-ins, conectarse a Amazon Web Services, Vagrant... y Docker. En este último caso, la implantación de Packer conducirá a la creación de una imagen Docker. De una determinada manera, es posible evitar la gramática `Dockerfile`, abriéndose a otras tecnologías.

En la actualidad, la herramienta Packer de Hashicorp no ha calado, aunque ya existía cuando se escribió la primera versión de este libro, en 2015. Sin embargo, es conveniente mantenerse atento sobre estos enfoques, porque las tecnologías innovadoras algunas veces renacen de sus cenizas.

## **4. INSTALACIÓN DE UN REGISTRO PRIVADO**

### **Primeros pasos para un registro privado**

El manejo de archivos de almacenamiento sigue un enfoque un poco simplista respecto a la sofisticación de un registro, incluso si el hecho de utilizar Docker Hub pudiera ser un impedimento por razones de confidencialidad. Afortunadamente, existe una solución ideal que acumula las ventajas de las dos soluciones: el registro privado.

Hemos visto anteriormente que era posible beneficiarse, en la oferta gratuita, de un almacén privado para una imagen en el registro Docker Hub, pero aquí hablamos de crear nuestro propio registro y por tanto, eliminar esta limitación. Las herramientas necesarias son gratuitas, el coste reside en la explotación y el alojamiento eventual del servidor.

Hay varias maneras de establecer un registro privado Docker, que vamos a detallar en las diferentes secciones siguientes.

### **1. Observación inicial importante**

Durante la primera edición del presente libro, este capítulo solo mostraba cómo establecer un registro privado utilizando sus propios medios, mencionando rápidamente la posible utilización de un servicio existente en el cloud. Durante la segunda edición, observamos que estas soluciones en el cloud están maduras y ofrecen un servicio extremadamente difícil y costoso de reproducir implementando un registro personal. Su alto grado de compartición permite reducir los costes de manera masiva. Delegar la seguridad en las plataformas especializadas en seguridad, le permite concentrarse en lo principal, a saber, la producción de las imágenes. Para terminar, ya no se presentan las cuestiones de actualización y mantenimiento en condiciones operativas, todo por un coste extremadamente bajo.

En resumen, las dos primeras secciones del presente capítulo se conservan porque permiten entender correctamente las implicaciones de un registro privado, incluso puede ayudar a trabajar en un contexto muy específico que necesite evitar los servicios cloud (proyecto de alto nivel de confidencialidad,

necesidad de utilización de un cloud privado, etc.). Pero en la gran mayoría de casos, establecer un registro privado consistirá en seguir las instrucciones de la tercera sección, que explica el concepto de registro privado en el cloud, utilizando como ejemplo el proporcionado por Microsoft en Azure.

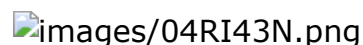
## 2. Advertencia sobre el antiguo registro

En primer lugar, tenga cuidado con no utilizar el registro antiguo. Hasta hace poco, el proyecto `docker-registry` (<https://github.com/docker/docker-registry>) se utilizaba para establecer un registro privado. Este proyecto, escrito en Python, se detiene en la versión 0.9.1. Ha salido una versión más reciente, completamente rescrita en Go (el lenguaje principal de desarrollo de Docker), y hace que el anterior registro haya quedado obsoleto. Esta versión se encuentra en <https://github.com/docker/distribution>.

La advertencia es necesaria porque este proyecto aparece en numerosos artículos y enlaces, así como en la primera página de resultados de una búsqueda sobre `docker private registry`. Al haber pasado dos años desde la primera edición del presente libro, puede haber dejado de pensar que esto siga sucediendo.

## 3. Imagen Docker en local

¿Qué hay más sencillo para desarrollar un servidor, que utilizar una imagen pública y arrancar un contenedor con ella? Esto es exactamente lo que vamos a hacer, utilizando la imagen oficial llamada `registry`, para crear nuestro propio servidor de registro. Una ventaja colateral es que la advertencia anterior sobre el carácter obsoleto del antiguo proyecto, es menos importante: en efecto, `library/registry` en Docker Hub, es el único almacén que contiene las versiones de `Dockerfile`, radicalmente diferentes entre la 0.9.1 y la 2.0. Además, para Docker Hub, las cosas están claras y solo se muestran desde ahora las versiones de la rama 2.0. La antigua versión ya no está disponible:




Hay alguna paradoja en utilizar el registro público para recuperar una imagen, que contiene el servidor necesario para la implantación del mismo registro en forma privada.

En la práctica, esto permite ahorrar esfuerzos de manera extraordinaria:

Escriba el siguiente comando:

```
docker run -d --name registro -p 88:5000 registry
```

Este sencillo comando es suficiente para arrancar un registro privado.

 Como es habitual, si es la primera vez que la máquina host utiliza esta imagen, será necesario un poco de tiempo para que se descargue. Las siguientes veces que se utilice, será extremadamente rápida.

En un contexto industrial, se hubiera podido realizar la elección explícita de una versión, pero aquí el enfoque es no especificar la versión y dejar que el comportamiento por defecto utilice la marcada por el tag `latest`. El objetivo de esta operación es recordar que la elección de una versión fija es, sin duda una promesa adicional de estabilidad, pero hay que pagar un precio que consiste en perder la evolución del producto, así como sus correcciones. Para un producto tan sensible para la seguridad como un registro, necesariamente abierto en Internet, puede resultar más sensato confiar en la compatibilidad ascendente para beneficiarse plenamente de las correcciones.

Un primer vistazo en un navegador que apunta al puerto 88 (o a cualquier otro puerto que haya seleccionado para exponer su registro), permite ver que el servidor está activo:

 [images/39.png](#)

La URL `/v2/` nos da acceso al API en versión 2 del registro y en ausencia de cualquier argumento o porción de URL adicional, el API solicitado devuelve una lista vacía, pero el hecho de que veamos los separadores de lista de JSON (los paréntesis) en la pestaña **Datos sin procesar** que proporciona Firefox, prueba que el servidor responde correctamente.

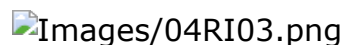
## 4. Apuntar a un registro dado

Para demostrar el uso del registro privado, vamos a depositar una imagen en este registro. Hemos visto recientemente el comando `push`, y vamos a añadir un sencillo argumento para dirigir la

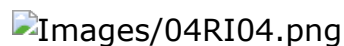
imagen deseada al registro privado, en lugar de al registro público Docker Hub.

Podría parecer lógico tener que conectarse a este nuevo registro y lanzar un nuevo comando `push` de una imagen de nombre existente, pero esta no es la manera en la que funciona Docker. Cuando hemos llamado a la imagen de ejemplo utilizada anteriormente, `jpgougoux/repeater`, este nombre completo cacheaba el servidor de registro por defecto, que resulta ser el registro público Docker Hub. Es sobre el que habíamos realizado una conexión, con la cuenta `jpgougoux`.

La imagen que hemos presentado hasta ahora en función del contenido necesario para arrancar un contenedor, es un poco más que esto, es decir, una asociación de este contenido con un almacén en un registro. Para enviar nuestra imagen al registro privado creado nuevamente, vamos a tener que comenzar por asignarle un nuevo nombre completo, gracias al comando `docker tag` y después únicamente, realizar la operación de `push` utilizando este nuevo nombre:



Si miramos los identificadores de las imágenes, efectivamente son similares a las dos denominaciones: Docker conserva el contenido común, tal y como es.



Volviendo a nuestro navegador y escribiendo una URL con el nombre de nuestra imagen, seguida de `/tags/list` vemos que la imagen se ha situado con sus tags. De esta manera confirmamos, su correcto funcionamiento:



La imagen ahora se almacena en un registro, desde donde usted puede recuperarla en teoría desde cualquier otra máquina de la red, salvo que nuestro ejemplo utiliza por el momento `localhost`, lo que impide que el cliente apunte a la máquina correcta. La siguiente sección realiza una implantación un poco más realista, pero el objetivo era comenzar con un ejemplo sencillo para mostrar que el registro se puede utilizar de manera muy sencilla.

Un rápido vistazo a los logs siempre es muy instructivo:

```
jpg@VM-UBUNTULTS:~/Docker/repeater$ docker logs registre

time="2017-10-03T09:19:42Z" level=warning msg="No HTTP secret
provided
- generated random secret. This may cause problems with uploads if
multiple
registries are behind a load-balancer. To provide a shared secret,
fill in
http.secret in the configuration file or set the REGISTRY_HTTP_SECRET
environment variable." go.version=go1.7.6
instance.id=e1525elf-ca2a-4488-a970-e4d3315eda32 version=v2.6.2

time="2017-10-03T09:19:42Z" level=info msg="redis not configured"
go.version=go1.7.6 instance.id=e1525elf-ca2a-4488-a970-e4d3315eda32
version=v2.6.2

(...)

172.17.0.1 - - [03/Oct/2017:09:56:37 +0000] "PUT
/v2/repeater/blobs/uploads/dc02ccdf-a570-4e82-9edc-c6f72f1e0e3e?
_state=
_lwO6qo0MIRt6aOB3WDKzKaeK8c4hQZEGZbEXlrW_lYp7Ik5hbWUiOiJyZXBl
dG10ZXVyIiwiaWV
VVRJCI6ImRjMDJjY2RmLWE1NzAtNGU4Mi05ZWVjLWM2ZjcyZjFlMGUzZSIs
Ik9mZnNldCI6NT
A5NywiU3RhcjRlZEF0IjoimjAxNy0xMC0wMlQwOT01NjozNloifQ%3D%3D&digest=
sha256%3A513bfe829a844cd871dda59159c581a1c51dcf3e78a9bcf6cc398df29564f
e4e
HTTP/1.1" 201 0 "" "docker/17.04.0-ce go/go1.7.5 git-commit/4845c56
kernel/
4.4.0-75-generic os/linux arch/amd64 UpstreamClient(Docker-
Client/17.04.
0-ce \\(linux\\))"

(...)

172.17.0.1 - - [03/Oct/2017:09:56:37 +0000] "PUT /v2/repeater/
manifests/latest HTTP/1.1" 201 0 "" "docker/17.04.0-ce go/go1.7.5
git-commit/4845c56 kernel/4.4.0-75-generic os/linux arch/amd64
UpstreamClient(Docker-Client/17.04.0-ce \\(linux\\))"

(...)

172.17.0.1 - - [03/Oct/2017:09:56:55 +0000] "GET
/v2/repeater/tags/list
HTTP/1.1" 200 40 "" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
rv:53.0) Gecko/20100101 Firefox/53.0"
```

Vemos que se muestra sucesivamente el mensaje del registro que indica que la seguridad por certificado no se ha configurado (con una explicación de bienvenida que indica cómo hacerlo). El mensaje explica que no se ha encontrado ninguna caché Redis para mejorar el rendimiento (pero el registro sabe cómo seguir sin esto). Una llamada al almacén de un blob (*Binary Large Object*), sin duda se corresponde con una parte de la imagen `repeater` que hemos enviado al registro privado. Otra se corresponde a priori con el registro de la etiqueta en esta misma imagen. Para terminar, la línea correspondiente a la última llamada realizada en el ejemplo, comprueba que la operación `docker push` de envío de la imagen ha recuperado correctamente el resultado.

## 5. Registro en una red pública

Como hemos comprobado al final de la sección anterior, es poco lógico establecer un registro en la máquina host, dando por hecho que esta tiene ya las imágenes. El objetivo del registro es favorecer la compartición, por lo que su ubicación lógicamente debe ser en un servidor accesible, público (porque en este caso, el registro público Docker Hub se utiliza mucho), al menos para todas las personas a las que la persona que prepara las imágenes desee ponerlas a su disposición.

Este servidor podría ser un servidor central en una red de empresa, un servidor securizado en cloud o cualquier otra máquina accesible a todo el grupo de personas destino, como consumidores. En un primer momento, vamos a mostrar cómo instalar un registro en una red interna. Este enfoque es un poco más ligero, en la medida en que nos permite permanecer en un entorno de confianza y por tanto, desenganchar el sistema de encriptado de los datos transferidos, y esto sin correr demasiados riesgos.

Por lo tanto, si desde el primer ejemplo hubiéramos utilizado una máquina remota, hubiéramos recibido un mensaje de error de tipo `server gave HTTP response to HTTPS client`. Debido al hecho de que, por defecto, el demonio Docker solo acepta hablar localmente con un registro no seguro. Para contrarrestar esto, un enfoque rápido pero menos seguro consiste en añadir en las opciones del demonio el registro a autorizar.

Por ejemplo:

En la máquina desde la que desea acceder al registro, edite el archivo `/etc/docker/daemon.json`. Sin duda será




necesario utilizar el prefijo del comando `sudo`. Es necesario elevar los permisos para modificar este archivo.

Añada la línea `{ "insecure-registries": ["swarmjpg.westeurope.cloudapp.azure.com:88"] }`, sustituyendo la URL por la de su registro.

Guarde el archivo.

Lance el comando `sudo systemctl restart docker` para volver a arrancar el demonio Docker de manera que tenga en cuenta el cambio de configuración.

 Aunque parezca evidente, preste atención al hecho de que detener el demonio provocará la parada de todos los contenedores lanzados en su interior. Este recordatorio se debe al hecho de que el autor, para simular un registro remoto, utilizó una dirección DNS apuntando a la máquina local. De repente, durante el cambio de configuración en el cliente, el servidor que en la práctica era la misma máquina, también se puso a cero y el registro ya no respondía naturalmente, generando lógicamente la aparición de un mensaje "getsockopt connection refused".

## **a. Escenario y preparación de las máquinas**

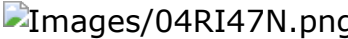
El escenario que vamos a establecer, se resume en el siguiente diagrama:

 Images/esquema120.png

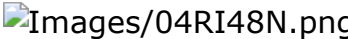
Tres máquinas en un cloud que forman parte de la misma red 10.0.1.\*, van a servir respectivamente de infraestructura para el registro (Node1), de cliente depositario de una imagen (Node2, en la interacción numerada con 1) y de cliente que recuperan esta misma imagen con destino a su caché local de imágenes Docker (Node3, en la interacción anotada como 2). La etapa numerada con 0 y que se explicará en la sección inmediatamente siguiente, será la implementación del registro. La primera máquina que se beneficia de una entrada DNS, lo que permitirá al resto no tener que utilizar una dirección IP en duro. La otra ventaja es que exponiendo el flujo en Internet, veremos los eventuales comandos que hay que establecer para abrir los puertos, dar seguridad al acceso, etc.

## **b. Puesta en marcha del registro**

La primera etapa consiste en arrancar el registro privado en la máquina elegida. No hay diferencia respecto al enfoque "todo en local", mostrado hasta ahora. Se trata de la misma línea de comandos, excepto que esta vez utilizaremos el puerto 5000 en lugar del puerto 88 (explicaremos más adelante esta elección):

Images/04RI47N.png

Un vistazo con el comando `docker ps` permite validar que el registro se ha lanzado correctamente:

Images/04RI48N.png

Dando por hecho que hemos expuesto el puerto 5000, conviene intentar acceder para gestionar las eventuales reglas de cortafuegos y otros aspectos de seguridad de red, que podrían impedir el acceso a la máquina Node1 desde el exterior, utilizando su nombre completo DNS. En la infraestructura utilizada por el autor, a saber tres máquinas en el cloud Azure, la operación consiste en añadir un ajuste de seguridad al tráfico entrante en el recurso node1-nsg (para Network Security Group).

images/71.png

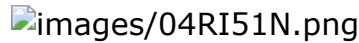
El botón **Agregar** permite acceder a una interfaz de creación de un ajuste, validando con el botón **OK**. La aplicación es muy rápida (algunos segundos en general, pero atención, la disponibilidad real del puerto necesita algunos segundos adicionales después de que el portal Azure haya indicado el éxito de la modificación).

images/72.png

Ahora nos podemos desconectar de esta primera máquina y conectarnos a la que vamos a utilizar para enviar la imagen elegida al registro (en nuestro caso, la máquina llamada Node2).

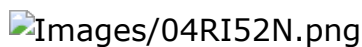
### c. Depositar la imagen desde otra máquina

El primer comando a lanzar desde la máquina Node2, es un acceso al registro por su interfaz HTTP. Los registros disponen de un API, como habíamos mostrado anteriormente. La siguiente llamada muestra que el registro responde bien:

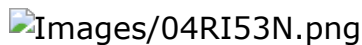


La lectura no es sencilla porque `curl` no pasa a la línea anterior en la visualización de la línea de comandos, pero se ve claramente que la llamada devuelve un mensaje que contiene dos paréntesis, correspondientes a un contenido JSON vacío.

Una vez aclarado esto, la operación es la misma que antes, saber utilizar `tag` para dar un nuevo nombre a la imagen (comprobamos en la siguiente captura que esta acción ya se ha realizado, listando las imágenes) y después, utilizar el comando `push` para enviar esta imagen al registro.



Sin embargo, si el comando `docker push` se lanza de manera inmediata, se genera un mensaje de error igual que el descrito anteriormente:



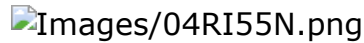
En efecto, el registro no estaba securizado por los certificados y un intercambio HTTPS, por lo que el demonio Docker local rechaza por defecto la conexión. Vamos a realizar la modificación necesaria en la configuración, para autorizarla (evidentemente, en una arquitectura de producción realizaremos la configuración completa de seguridad, siguiendo la documentación Docker):

```
Edite el archivo /etc/docker/daemon.json de la máquina Node2,
añadiéndole la línea {"insecure-registries":
["swarmjpg.westeurope.cloudapp.azure.com:5000"] }.
```

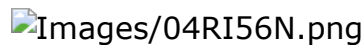
Guarde el archivo.

Lance el comando `sudo systemctl reload docker` para que tenga en cuenta el cambio de configuración (este método es menos brusco que un comando `restart`).

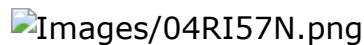
Al final de la carga, la interfaz devuelve un resumen concentrado de seguridad. Desde algunas versiones anteriores, el mecanismo de intercambio de las imágenes por Docker estaba securizado contra las modificaciones sobre la marcha de imágenes, por parte de un tercero no seguro. El resumen de seguridad condensado permite comprobar que la imagen recibida, es estrictamente idéntica a la enviada.



Si la imagen ya se ha metido en el registro, estos mensajes se acompañan de una gran rapidez de ejecución del comando. Docker detecta que no es necesaria ninguna modificación, evitando de esta manera una sobrecarga inútil de la red. Además, solo se enviarán las capas que hayan cambiado si se ha modificado `Dockerfile` y la imagen se reconstruye en consecuencia, como se ilustra en la siguiente captura de pantalla:



Para validar que el comando de envío ha alcanzado correctamente su destino, podemos utilizar de nuevo el API, situándonos esta vez en una máquina cualquiera con acceso a Internet. Con el objetivo de variar los enfoques, mostramos esta vez el resultado en un navegador gráfico (Microsoft Edge), en lugar de en una línea de comandos (`curl`):

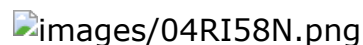


#### **d. Utilización de la imagen desde una tercera máquina**

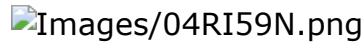
Una vez que se ha depositado la imagen en el registro, ahora vamos a dar una vuelta más en la demostración, consumiendo desde una tercera máquina de la red en este caso de la máquina Node3.

Como sucede con Node2, en primer lugar conviene añadir el registro externo a la lista de registros no seguros aportado por el archivo `/etc/docker/daemon.json`. No ilustramos de nuevo el comando, que es estrictamente idéntico al utilizado sobre Node2.

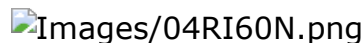
En primer lugar, se verifica no solo que la imagen ya no está presente, sino que el registro está vacío:



A continuación, en lugar de realizar una operación `docker pull`, lanzamos directamente un comando `docker run`, que es más habitual. El demonio se encarga de llevar la imagen a la caché local si es necesario, indicando el progreso de la descarga:



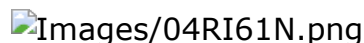
Una vez terminada, es decir, al cabo de algunos minutos, si la imagen básica Ubuntu se debe descargar y usted utiliza una conexión Internet personal o bien al cabo de algunos segundos si utiliza una máquina en el cloud, que se beneficiará de un ancho de banda enorme y además de un nivel de cacheado intermedio, la imagen se instancia en forma de contenedor que reproduce el mecanismo de heartbeat que nos ha servido de ejemplo hasta ahora:



### **e. Eliminación de la imagen en la máquina origen**

Lógicamente, una vez que la máquina Node2 ha enviado una imagen al registro central, podemos eliminar esta imagen de la caché local. Ya hemos visto el comando `rmi`, pero nuestro ejemplo nos permite profundizar en su comportamiento.

La siguiente captura de pantalla muestra el encadenamiento de comandos necesarios para limpiar definitivamente las imágenes:



El detalle de los comandos y de los resultados es instructivo:

- La lista de las imágenes hace que aparezcan lógicamente los dos tags para el mismo identificador de imagen: se trata de la misma imagen Docker a la que simplemente hemos asociado un nombre, para situarla en el registro privado.
- Utilizando el comando `rmi` con el identificador en lugar del nombre, expresamos la voluntad de eliminar todos los tags, pero un mensaje explica que, como el tag está asociado a múltiples almacenes (el local y el del registro), es necesario utilizar la opción `-f`.
- La siguiente llamada con esta opción `-f`, elimina de manera forzada la imagen y todas sus etiquetas.
- Otra llamada al comando `docker images` permite validar que las dos etiquetas se han eliminado, así como las imágenes correspondientes en sentido de los identificadores hexadecimales.



# Un registro más profesional

## 1. Gestión de la persistencia

### a. Gestión local por volumen

El hecho de arrancar el registro en una máquina diferente de la que consume las imágenes, implicó una primera etapa en el despliegue de un registro realista, pero por el momento estamos lejos tener un registro listo para una utilización en producción.

El primer problema que se plantea es que, por el momento, el almacenamiento de las imágenes situadas en el registro se hace en el contenedor Docker en sí mismo. Desde este momento, ¿qué pasa si el contenedor se detiene y se olvida validar su estado en una nueva imagen? Simplemente, el contenido del registro se pierde. Es evidente que este tipo de comportamiento es inimaginable para un registro en producción y por lo tanto, ahora vamos a centrarnos en gestionarlo de manera más sostenible.

Un primer enfoque un poco simplista, sin lugar a dudas, pero funcional, consiste en utilizar los volúmenes para redirigir al lugar donde el contenedor del registro almacena las imágenes que se le envían, a una ubicación un poco más permanente por ejemplo, un directorio de la máquina host sobre la que se haga un backup regularmente, incluso se sincronice de manera permanente con una máquina remota. En un contexto más profesional, el destino podría ser una SAN bay, compartición NFS (*Network File System*), etc.


La documentación explica que el comportamiento por defecto del registro que Docker ofrece, es almacenar en `/tmp/registry-dev`. Por lo tanto, sustituimos este directorio por otro.

Lance una nueva instancia de registro con el mismo comando que antes, pero añadiendo una opción de gestión de volumen (argumento que empieza por `-v` en el siguiente ejemplo), de manera que el almacenamiento del registro se realice en un directorio de su conveniencia:

```
docker run -d --name register -p 5000:5000 -v  
/home/jpg/miregistro/contenido:/tmp/registry-dev  
jpgougoux/registry-volume
```

Desde la máquina que haya servido para esta operación, lance un comando `push` para enviar el contenido al registro.

Una vez terminado, puede comprobar el resultado de este comando, observando el contenido del directorio elegido para el almacenamiento del registro:

 Images/280.png

La arborescencia muestra el nombre de la imagen en la que hemos realizado la operación de push, en este caso `repeater`, en el directorio `repositories`.

## **b. Observación sur SELinux**

SELinux, para *Security Enhanced Linux*, es un módulo de seguridad integrado en Linux, que permite definir las políticas de permisos de acceso a los diferentes recursos del sistema de manera muy fina.

En función de su configuración, SELinux puede bloquear la escritura a través del proceso del registro en el directorio proporcionado por el mecanismo de los volúmenes, fuera de cualquier problema de permisos de escritura en el directorio en sí mismo. Entonces, en la máquina que aloja el contenedor de registro obtendremos un aviso SELinux, cuando lance el comando `push` desde otra máquina. En esta máquina, el error se devolverá como sigue o al menos, de manera parecida:

```
989122ba5665: Image push failed
FATA[0000] Error pushing to registry: Server error: unexpected
500 response status trying to initiate upload of repeater
jpg@Ubuntu-VirtualBox:~$ sudo docker push
miregistro:5000/repeater
The push refers to a repository [miregistro:5000/repeater]
(len: 1)
989122ba5665: Image push failed
```

En un enfoque de aprendizaje, puede ser suficiente pasar SELinux en modo permisivo (ver la documentación de su distribución para conocer la manera de proceder) o desactivarlo temporalmente. Esto permite comprobar que la configuración funciona aparte de cualquier capa superpuesta de protección, después de ajustar los argumentos para que continúe funcionando una vez que se vuelve a implementar la seguridad, sin correr el riesgo de confusión intentando tratar las dos al mismo tiempo.



Evidentemente, en un contexto de producción quizás prefiera pedir a su administrador que configure correctamente el SELinux, para que deje pasar este tipo de llamadas, además de los permisos existentes. No vamos a desarrollar aquí los diferentes métodos posibles, el tema está relacionado y necesita potencialmente una operación larga si queremos esperar el nivel de comprensión necesario para no comprometer la seguridad del sistema.

### c. Un almacenamiento más seguro

La operación mencionada más atrás, ha permitido poner el almacenamiento del registro en un lugar más seguro que un directorio gestionado por el contenedor en sí mismo y por lo tanto, expuesto a una parada imprevista. En una aplicación real, estaríamos todavía al inicio del trabajo, porque también sería necesario dar seguridad a este directorio, pensar en hacer una copia de seguridad, comprobar que el espacio en disco es suficiente, incluso realizar un montaje un poco más sofisticado para poder extenderlo dinámicamente, etc.

Por supuesto, todas estas operaciones están al alcance de cualquier administrador pero, ¿por qué en el cloud no se delega en empresas especializadas todas estas operaciones, que no son necesarias en nuestro núcleo de negocio? Observamos que el registro proporcionado por Docker se ha diseñado para utilizar de manera muy sencilla, los sistemas de almacenamiento remotos.

Es posible la configuración del registro, gracias a un archivo en formato YAML. El formato YAML (*Yet Another Markup Language*) es un lenguaje textual etiquetado, como XML o JSON. A continuación se muestra la forma por defecto de este archivo:

```
version: 0.1
log:
  level: debug
  fields:
    service: registry
    environment: development
storage:
  cache:
    layerinfo: inmemory
  filesystem:
    rootdirectory: /tmp/registry-dev
http:
  addr: :5000
  secret: asecretforlocaldevelopment
  debug:
    addr: localhost:5001
redis:
  addr: localhost:6379
```

```
pool:
  maxidle: 16
  maxactive: 64
  idletimeout: 300s
  dialtimeout: 10ms
  readtimeout: 10ms
  writetimeout: 10ms
notifications:
  endpoints:
    - name: local-8082
      url: http://localhost:5003/callback
      headers:
        Authorization: [Bearer <an example token>]
      timeout: 1s
      threshold: 10
      backoff: 1s
      disabled: true
    - name: local-8083
      url: http://localhost:8083/callback
      timeout: 1s
      threshold: 10
      backoff: 1s
      disabled: true
```

Una entrada en particular nos va a permitir jugar con el almacenamiento del registro, a saber, la sección `storage`. Un vistazo a una página de la documentación relacionada (<https://docs.docker.com/registry/configuration/>), nos enseña que el comportamiento del almacenamiento se puede ajustar de manera que apunte a una cuenta de almacenamiento Azure (también está disponible una opción para utilizar una unidad de almacenamiento (bucket) Amazon S3).

Con el objetivo de no alejarnos demasiado de nuestro objetivo de entender correctamente los mecanismos de Docker, no mostraremos en detalle los argumentos necesarios, sino simplemente llamamos la atención del lector sobre algunos puntos importantes.

En primer lugar, el almacenamiento se debe realizar lo más cerca posible del registro, porque las imágenes que pesan mucho y los intercambios, pueden consumir mucho ancho de banda. De esta manera, si su registro se aloja en Azure, sobre todo no hay que utilizar un almacenamiento en Amazon o a la inversa.

Otro punto importante es dar seguridad a las claves de acceso a sus cuentas de almacenamiento y, en particular, jamás escribirlas en los archivos de configuración. Es más seguro utilizar argumentos a los que se les pase el valor con ayuda de variables de entorno en los scripts de puesta en marcha de su registro. De esta manera, no solo los códigos secretos no estarán en los archivos de configuración del registro, sino que además podrá evitar cualquier visualización de la contraseña durante las operaciones.

## 2. Securitización del registro

Por defecto, el registro no gestiona autorizaciones y deja que cualquiera lo utilice. La configuración del archivo de configuración permite remediar esto y establecer una autorización fundada en tokens. El método es particularmente adecuado si desea delegar la autenticación a un tercero externo.

Sin embargo, en nuestro enfoque, que consiste en un registro privado, puede parecer más lógico utilizar un método alternativo, a saber, confiar la autenticación a un servidor web ubicado en el front del registro. Este método también tiene la ventaja de permitir aprovechar toda la riqueza funcional de un servidor web real (redirección, filtrado, enrutamiento, seguridad, etc.), y es precisamente esto lo que vamos a mostrar.

### a. Preparación de las claves para el canal encriptado

El registro Docker al que vamos a dar seguridad, utiliza TLS/SSL para cifrar los datos intercambiados. Para esto, es necesario comenzar por una etapa de generación de un certificado, así como de una clave que será firmada por este certificado. Aunque esta etapa no esté particularmente relacionada con Docker, detallarla es instructivo respecto al uso que vamos a hacer a continuación de los archivos generados.

Todas las operaciones siguientes, se hacen como administrador. Por lo tanto, es necesario utilizar el prefijo `sudo` o bien lanzar un comando `sudo -s` para abrir un shell en modo administrador. La segunda solución se considera más peligrosa porque una manipulación incorrecta puede tener consecuencias nefastas, pero tener el hábito de escribir `sudo` al inicio de cada línea, tampoco está exento de riesgos.

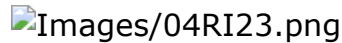
Sitúese en el directorio `/etc/ssl/certs`.

Cree una clave y su certificado X509 para la autoridad de certificación, de tamaño 2.048 bits y válido para dos años, utilizando el comando `openssl` como se ilustra a continuación, ajustando los nombres de archivos deseados.

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout gouigoux-CA.key  
-days 730 -out gouigoux-CA.crt
```

Responda a las diferentes preguntas que plantea el programa, con sus características propias (los valores efectivos importan

poco en lo que resta del ejemplo):



A continuación cree una segunda clave, que esta vez servirá para la implantación de TLS/SSL en el registro Docker:

```
openssl genrsa -out registredocker.key 2048
```

A partir de esta clave, genere una petición de firma:

```
openssl req -new -key registredocker.key -out registredocker.csr
```

De nuevo, responda a las preguntas con sus características propias. Esta vez, un valor es particularmente importante, a saber el Common Name: es necesario que sea el nombre del servidor que va a proteger. En nuestro ejemplo, se trata de miregistro pero, si estuviéramos en una exposición externa real, sería necesario poner el FQDN (*Fully Qualified Domain Name*, por ejemplo, `docker.eni.es`). Si hemos dado seguridad a la misma infraestructura utilizada en el ejemplo anterior, se deberá indicar `swarmjpg.westeurope.cloudapp.azure.com`.



La siguiente etapa consiste en utilizar la autoridad de certificación que habíamos creado al inicio de esta operación, para firmar la petición de clave para la máquina que expone el registro. El certificado creado de esta manera, dura un año.

```
openssl x509 -req -in registredocker.csr -CA gouigoux-CA.crt  
-CAkey gouigoux-CA.key -CAcreateserial -out registredocker.crt  
-days 365
```



Por el momento, hemos creado todo en el directorio `/etc/ssl/certs`, pero la ubicación de las claves es `/etc/ssl/private`. Muévalas con ayuda del comando `mv`.

Los archivos `registredocker.srl` y `registredocker.csr` se pueden eliminar y no servirán en nuestro ejemplo.

## b. Implantación del front de Nginx

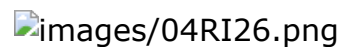
Estando listas las claves, podemos pasar a la implementación del servidor web que va a servir de parte front a nuestro registro

Docker. Para nuestro ejemplo, utilizaremos Nginx, pero las funcionalidades implantadas se pueden hacer con ayuda de un servidor Apache, IIS, u otro. Empezamos por la instalación propiamente dicha del servidor Nginx:

Escriba el comando `apt-get install nginx` (o el equivalente para otros sistemas, como `yum install nginx`).

Normalmente, al final de la instalación, ya está lanzado el servidor. Si este no es el caso, ejecute el comando `servicio nginx start`.

Lance un navegador sobre `http://localhost` para comprobar que Nginx funciona correctamente. Normalmente debe ver una pantalla similar a la siguiente:



La etapa siguiente consiste en modificar el archivo de configuración para enseñar a Nginx la manera en la que deseamos que se comporte como front del registro Docker.

Edite el archivo `/etc/nginx/conf.d/nginx.conf` y modifíquelo de manera que tenga el siguiente contenido:

```
server {
    listen 443 ssl;
    server_name _;

    ssl_certificate /etc/ssl/certs/registredocker.crt;
    ssl_certificate_key /etc/ssl/private/registredocker.key;

    client_max_body_size 0; # disable any limits to avoid HTTP 413
    for large image uploads

    proxy_set_header    Host $host;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    X-Real-IP $remote_addr;
    proxy_set_header    X-Forwarded-Proto $scheme;
    proxy_set_header    X-Original-URI $request_uri;
    proxy_set_header    Docker-Distribution-API-Version registry/2.0;
    proxy_set_header    Autorización "";
    proxy_read_timeout  900;


    location / {
        proxy_pass http://miregistro:5000;
    }
}
```

Registre las modificaciones.

A continuación, se muestra el detalle funcional línea a línea, del archivo:

- `listen 443 ssl`: el servidor Nginx sabe que debe escuchar en el puerto 443, en modo SSL (es decir utilizando el canal HTTP cifrado). Puede utilizar otro puerto diferente del 443, pero este número de puerto normalmente se utiliza para HTTPS.
- `server name _`: Nginx escuchará todas las direcciones entrantes, sea cual sea la gramática utilizada (`localhost`, `miregistro` o cualquier otro alias del servidor). También hubiéramos podido utilizar `server name miregistro`.
- Las dos líneas siguientes indican el certificado y la clave a utilizar para establecer la comunicación segura. Es aquí donde proporcionamos los archivos creados anteriormente y, en particular, el certificado firmado por la autoridad de certificación que habíamos implementado.
- `cliente max_body_size 0`: desactiva el límite de tamaño de los contenidos, lo que es necesario para poder hacer pasar las imágenes, con un tamaño varias veces superior. Sin este argumento, corre el riesgo de sufrir errores de tipo HTTP 413.
- `proxy_set_header`: estos comandos añaden los encabezados (*headers*) en los mensajes. No todos son necesarios en todos los casos, pero cada uno tiene un uso para una versión o un contexto de uso dado. No es imposible que durante su utilización, un header dado deje de ser imprescindible porque configura una solución que ya no es necesaria. Por ejemplo:
  - `Host` es necesario para el cliente Docker según la documentación oficial, pero en la práctica, su ausencia no parece molestar en una versión 1.6.
  - `Authorization ""` se utiliza para solucionar el error registrado en [https://github.com/docker-registry/issues/170](https://github.com/docker/docker-registry/issues/170). Es posible que este truco deje de ser necesario en cualquier momento.
  - `Docker-Distribution-API-Version registry/2.0`: sirve para decir a la llamada que se trata de la versión 2.0 del API del registro.

- `location /:` establece el comportamiento que el servidor Nginx debe tener para todas las llamadas, en ausencia de otra sección que especializa este comportamiento para una ruta más concreta en la URL.
- `proxy_pass:` indica que este comportamiento es justamente enrutar el contenido a `miregistro:5000`, que es nuestro registro Docker arrancado anteriormente (atención, si usa un proxy Nginx no va a volver pasar más por la dirección DNS, por lo que en la infraestructura del ejemplo anterior, no hubiéramos sustituido este valor por `swarmjpg.westeurope.cloudapp.azure.com` sino por `10.0.0.1`).


 Respecto a las especializaciones de comportamiento para algunas rutas, encontrará en la documentación Docker (en particular en <https://github.com/docker/docker-registry/tree/master/contrib/nginx>), ejemplos de archivos de configuración Nginx que explican que el comando de ping del registro se debe enrutar de manera específica. Esto se aplica al antiguo registro (versión 0.9), pero ahora parece inútil para la versión 2.0.

Pruebe que no haya introducido errores en el archivo, utilizando el comando `nginx -t`, que debe devolver un mensaje como el siguiente:

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Si este no es el caso, corrija los errores que haya dado la prueba.

Recargue la configuración de Nginx gracias al comando `service nginx reload`.

 Es inútil volver a arrancar completamente Nginx con un servicio `nginx restart` tan pronto como modifique simplemente la configuración. La operación `reload` es más rápida y menos impactante.

Compruebe que el comando provoca un mensaje [OK]:

 `images/57.png`

Si no es el caso y ve `[fail]` como mensaje, vuelva al comando de diagnóstico anterior para encontrar el problema

(en esta etapa donde el servidor todavía no se ha lanzado, los logs de `/var/log/nginx` no son de gran utilidad y es el comando de prueba del archivo de configuración el que normalmente le mostrará el problema).

Acceda ahora usando HTTPS a la máquina `miregistro` (que es la misma que `localhost`, porque hemos ubicado el front web en la misma máquina que el contenedor Docker, que no será probablemente el caso en un entorno de producción).

En un primer momento, utilice simplemente un navegador web. Debería obtener una visualización parecida a la siguiente:



La advertencia es lógica, teniendo en cuenta que no hemos utilizado certificados reales reconocidos por el sistema como parte de una cadena de confianza.

Pulse en **Entiendo los riesgos**.

Debería aparecer una ventana como la siguiente (eventualmente sin el mensaje "Sitio no seguro " si su Common Name se ha configurado correctamente, que no es el caso en la siguiente captura de pantalla).



Pulse en **Obtener certificado y Ver...** para validar que se trata del certificado que acaba de crear (un buen administrador nunca es suficientemente confiado).

Si es el caso, pulse en el botón **Confirmar excepción de seguridad**, después de haber marcado **Guardar esta excepción de manera permanente** si no quiere que esta advertencia aparezca en cada uso.

El navegador tiene en cuenta su aceptación y relanza la carga, que debería provocar el siguiente contenido:



En esta etapa de nuestro ejemplo, si no ve un contenido puede ser porque no haya lanzado un contenedor en la imagen `registry:2.0`, o bien porque el puerto no se haya mapeado en el puerto 5000 de la máquina `miregistro`, incluso porque ya no tiene la imagen `repeater` en este registro, como consecuencia



de una operación anterior. Tómese tiempo para volver atrás en el capítulo para poner a punto esta etapa antes de continuar, para no correr el riesgo de tener que buscar en una configuración dada los errores que provienen de otra parte y perder su tiempo de esta manera.

### c. Acceso por el cliente Docker

Llegados a este punto, el servidor Nginx hace más o menos correctamente el trabajo que le habíamos pedido que hiciera (excepto que es necesario aceptar una excepción en el certificado SSL). Pero si creamos un nuevo tag para una imagen en el registro `miregistro:443` e intentamos un comando `push`, obtenemos un error:



El error es muy explícito: Docker explica que intenta acceder a `miregistro:443`, pero que el HTTPS no es de un nivel suficiente para él. Por lo tanto, podemos declarar el registro como no seguro, pero esta es la solución que habíamos utilizado previamente, y objetivo de esta sección es establecer un registro "verdadero", por lo que vamos a coger el camino más elegante, es decir, declarar en Docker el certificado de autoridad necesario para validar la clave.

El mensaje de error nos guía:

Cree un directorio llamado `miregistro:443` en `/etc/docker/certs.d` (comando `mkdir`).

Copie en este directorio el certificado correspondiente a la autoridad de certificación con la que habíamos generado un poco antes la clave del registro, y que habíamos utilizado para firmarla. En nuestro ejemplo, el archivo se llama `gouigoux-CA.crt`. De paso, renómbrelo por `ca.crt`.

Estas dos operaciones se pueden realizar en un único comando:

```
cp /etc/ssl/certs/gouigoux-CA.crt  
/etc/docker/certs.d/miregistro:443/ca.crt
```

Vuelva a intentar el comando `push` (inútil volver a arrancar Docker, su comportamiento es dinámico desde este punto de vista).

El mensaje de error ha cambiado esta vez, lo que indica un progreso:

```
FATA[0000] Error pushing to registry: Put
https://miregistro/v2/repeater/blobs/uploads/3558fd00-161e-
4c37-9e2d-b2cb8ddd7e19?_state=2QRJaVv2x_D_6Y5Ksk1lwh-
FGNoR9RdfRLBx6YQHqhf7Ik5hbWUiOiJyZXBlcGl0ZXVvIiwivVVJRCl6IjM1NThm
ZDAwLTE2MWUtNGMzNy05ZTJkLWIyY2I4ZGRkN2UxOSIsIk9mZnNldCI6MCwiU3Rhc
nRlZEF0IjoimjAxNS0wNS0wOVQwOToxMDowMi42Mjc3NTU3OTJaIn0%3D&digest=
sha256%3Aa3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22
955b46d4: x509: certificate signed by unknown authority
```

Esta vez, Docker explica que el certificado no se ha firmado por una autoridad de certificación reconocida, pero no deja dar una excepción, como era posible con el navegador Firefox, anteriormente. La solución más limpia consiste en volver a añadir el certificado de autoridad de certificación en el almacén de certificados de la máquina, de manera que Docker considere que el certificado que se le presenta, es válido.

Posiciónese en `/usr/local/share/ca-certificates`.

Cree un directorio, que llamaremos `gouigoux-cert` en nuestro ejemplo.


Copie en este directorio el certificado de la autoridad de certificación, a saber, el archivo que se corresponde en nuestro ejemplo con `gouigoux-CA.crt`.

Lance el comando `update-ca-certificates`, que actualiza los certificados. Debe ver el mensaje `"1 added"`, que muestra que la nueva autoridad de certificación se ha tenido en cuenta.

Vuelva a lanzar Docker con el comando `servicedocker restart`. Contrariamente a cuando se añade la autoridad de certificación en su propio directorio, Docker no tiene un comportamiento dinámico respecto al almacén de certificados del sistema.

Vuelva a lanzar el comando `push`, que esta vez deberá funcionar correctamente.

Disponemos de un front web que securiza el flujo del registro Docker y funcional al mismo tiempo, para la llamada de los API en un navegador y para la comunicación por un cliente Docker 1.6.

 Durante la implantación del registro, anteriormente habíamos cambiado de puerto, para exponer el 5000 en lugar del 88. El objetivo era preparar el terreno a Nginx, que es el más indicado

para exponer los puertos 88 o 443, de manera tradicional asociado a una exposición HTTP. La utilización de un número superior a 1024 también tiene por objetivo evitar algunas limitaciones sobre la gestión de red de VirtualBox.

## **d. Herramientas adicionales de diagnóstico**

Los comandos anteriores son demasiado numerosos. Se han añadido numerosos detalles y explicaciones, de manera que se haga más robusto el proceso de implementación. Proporcionar el archivo de configuración correcto, no es útil en sí mismo, porque hay todas las opciones de que, sobre una configuración dada, un solo detalle impida el correcto funcionamiento del conjunto. Es mejor explicar para qué sirve cada línea y cada comando y dar las herramientas para validar el funcionamiento.

Esto es lo que se hizo antes, pero el lector encontrará ciertamente útil conocer algunos métodos adicionales de diagnóstico.

En primer lugar, los logs de Nginx están en `/var/log/nginx`. Hay disponibles un archivo `access.log` y un archivo `error.log`. Listan respectivamente todas las peticiones a Nginx con los códigos de retorno asociados (no los contenidos) y todos los errores encontrados durante su ejecución. El comando `tail` es práctico para ver las últimas líneas de estos archivos. Por ejemplo, una petición del segundo archivo podría mostrar el siguiente contenido:

images/60.png

La penúltima línea se corresponde con el error 413 de demasiado contenido, del que hemos hablado más atrás y que lleva a la utilización de la opción `client_max_body_size`, en el archivo de configuración.

Para esto, preste atención a los numerosos artículos disponibles en Internet, que explican que la opción `chunkin on` es obligatoria (<https://github.com/docker/docker/issues/1486>). Esto es cierto para la versión 0.9 del registro, pero ya no lo es para la versión 2.0.

También observamos que esto es para evitar ver este tipo de problema demasiado tarde, que no se había implementado en los

ejemplos anteriores con la imagen `hello-world`, que es extremadamente ligera y no hubiera provocado el error 413.

Siempre enlazando con Nginx, si obtiene un error SSL: `certificate subject name '...' does not match target host name '...'`, se trata de un error en la generación del segundo certificado, que se utiliza por Common Name el FQDN del servidor.

En los casos más extremos, también es posible lanzar el demonio Docker en sí mismo en modo debug y seguir sus trazas detalladas, pero lo que era una práctica habitual durante los primeros años, ya no lo es hoy ya que el producto es mucho más estable. Este modo de funcionamiento solo tiene sentido para los expertos que buscan entender los mecanismos internos de Docker.

## **e. Generación del archivo de los usuarios autorizados**

Realizada la primera etapa de securización del registro Docker, podemos pasar por un canal seguro de tipo TLS/SSL. Para afinar, podemos configurar Nginx para redirigir las llamadas HTTP a este canal, organizar la difusión de la autoridad de certificación o el uso de un certificado "oficial", etc. Pero esto no afecta más a Docker, mientras que la gestión de los usuarios está relacionada con él. Por lo tanto, este es el aspecto que vamos a establecer a través de Nginx.

Como se ha visto más atrás, es posible gestionar las autorizaciones de los usuarios directamente a nivel del registro Docker, por medio de un mecanismo de token, pero en la presente sección vamos a realizar las funciones de autenticación, identificación y autorización por medio de Nginx, por razones de separación propia de las responsabilidades.

En primer lugar, es necesario construir un archivo de usuarios. Por razones de simplicidad, elegiremos este enfoque en lugar de conectarnos a un directorio LDAP u otro backend. De nuevo una vez más, el interés de conectar estas funciones a nivel de Nginx, es que a continuación seremos más libres de hacerlas evolucionar a otros mecanismos. El objetivo es concentrarse en los enlaces con el registro Docker, no en la autenticación con Nginx en general.

Vamos a utilizar el programa `htpasswd` para formar un archivo de usuarios. Este programa forma parte del paquete `apache2-`

utils.

Escriba el comando `which htpasswd`.

Si la salida es `/usr/bin/htpasswd`, el programa ya está instalado en su sistema.

En caso contrario, instálelo con el comando `apt-get install apache2-utils` (o `yum install apache2-utils`, según su distribución Linux).


Ejecute el siguiente comando:

```
htpasswd -c /etc/nginx/registredocker.htpasswd usuario1
```

Asigne una contraseña como se pide.

Vuelva a comenzar la operación para otros si es necesario, omitiendo la opción `-c`, que era necesaria para crear el archivo.

El archivo que almacena los usuarios y las huellas (hash) de contraseñas, está listo para ser utilizado en Nginx.

 Tenga cuidado con no utilizar menos de cuatro letras para los identificadores de usuarios. Esto no plantea problemas con la formación del archivo, ni para Nginx, pero Docker rechazará utilizarlas para conectarse. Además, en función del contexto, el mensaje de error en la autenticación no será de los más claros...

## f. Añadir la autenticación en Nginx

Ahora podemos configurar Nginx para que utilice este archivo para bloquear el acceso no autorizado al registro que forma el proxy.

Edite el archivo `/etc/nginx/conf.d/nginx.conf`.

Modifique la sección `location /` añadiendo las dos líneas necesarias para que sean como la siguiente:

```
Location / {
    auth_basic          "Restricted";
    auth_basic_user_file registredocker.htpasswd;
    proxy_pass          http://miregistro:5000;
}
```

Ajuste el nombre del archivo en función del que haya creado.

Preste atención con no poner este archivo en otro lugar; Nginx necesita que esté presente en esta ubicación.

Pruebe la configuración con `nginx -t`. Trate los eventuales problemas (sintaxis incorrecta, por ejemplo).

Actualice la configuración de Nginx con servicio `nginx reload`. Compruebe que el mensaje es correcto [OK].

Recargue la página <https://miregistro/v2/repeater/tags/list> que nos ha servido de diagnóstico de funcionamiento hasta ahora. El resultado debería ser una solicitud de autenticación como la siguiente:



Escriba un nombre de usuario y una contraseña como las definidas durante la construcción del archivo por `htpasswd`.

Valide que el servidor devuelve el mismo resultado que anteriormente.



En caso de problema, compruebe en primer lugar que el proceso `docker` funciona siempre y que su contenedor `registro` escucha siempre en el puerto 5000.

Pruebe ahora con el mismo comando `push` habitual. Esta vez, debe ver una petición de conexión como la siguiente:



Conéctese.

Observe que Docker le pide además una dirección de e-mail.

A continuación, el comando se debería desarrollar como habitualmente, pero puede aparecer un error como este:

```
ERRO[1165] unable to login against registry endpoint
https://miregistro:443/v1/: Login: 404 page not found
(Code: 404; Headers: map[Connection:[keep-alive] Docker-
Distribution-API-Version:[registry/2.0] Server:[nginx/1.6.2
(Ubuntu)] Date:[Sat, 09 May 2015 16:45:09 GMT] Content-
Type:[text/plain; charset=utf-8] Content-Length:[19]])
Login: 404 page not found
```

## g. Corrección en los headers

Una operación sutil requiere agregar una opción en el archivo de configuración de Nginx. En efecto, como se explica en <https://github.com/docker/distribution/issues/397>, el hecho de utilizar `proxy_set_header Docker-Distribution-API-Version registry/2.0;` para declarar que el registro funciona en 2.0, no es suficiente cuando active el modo Basic Authentication, como hemos hecho. Por lo tanto, es necesario utilizar un comando adicional:

Edite el archivo `/etc/nginx/conf.d/nginx.conf`.

Dentro de `server_name _;`, añada la siguiente línea:

```
more_set_headers 'Docker-Distribution-API-Version: registry/2.0';
```

Guarde sus modificaciones.


Intente una actualización de la configuración de Nginx con `service nginx reload`, y compruebe el error `[fail]`.

`more_set_headers` no es una funcionalidad básica de Nginx. Es necesario instalar un paquete adicional.

Lance `sudo apt-get install nginx-extras`.

Intente de nuevo cargar la configuración, lo que ahora debería pasar sin problema y por tanto, con un mensaje `[OK]`. Si este no es el caso, utilice `nginx -t` para diagnosticar.

Vuelva a lanzar el comando `push`, para validar que todo funciona.

 Observe que si utiliza una versión particularmente reciente de Nginx (1.7.5 al menos), puede pasar del paquete `nginx-extras` y sustituir la línea `more_set_headers 'Docker-Distribution-API-Version: registry/2.0';` por `add_header 'Docker-Distribution-API-Version: registry/2.0' always;`, la palabra clave `always` provoca el mismo comportamiento, a saber, enviar sistemáticamente el encabezado.

## **h. Gestión de la conexión**

Es posible conectarse directamente con el comando `docker login [registro]`, sin esperar a que un comando operativo lo haga necesario. Esto permite pasar las entradas en la línea de

comandos, incluso la contraseña si se desea, aunque no sea recomendable por razones de confidencialidad:

```
sudo docker login -u usuario1 -e user1@test.es  
miregistro:443
```

Si utiliza el comando `logout`, no olvide indicar el registro como argumento, en caso contrario, se desconectará de Docker Hub (que es el registro por defecto para el cliente Docker), y no del registro privado al que usted apunta.


```
sudo docker logout miregistro:443
```

En el momento de la escritura de este libro, no es posible modificar el registro por defecto utilizado por el cliente Docker, que apunta al índice Docker Hub. Esto proviene de una voluntad de la empresa Docker Inc. de favorecer al máximo la compartición de las imágenes que se erige como una parte importante de la filosofía del producto. No hay duda de que las empresas que exploten industrialmente Docker, habrán utilizado una redirección DNS para apuntar a su propio registro, incluso habrán modificado el código fuente por esta necesidad.

### 3. Un verdadero registro en producción

En las páginas anteriores, no habíamos agotado todas las opciones del archivo de configuración del registro. No vamos a explorar en detalle todas las sutilidades, pero listaremos a continuación las principales posibilidades ofrecidas:

- Utilización de un servidor Redis para la caché, en lugar que una sencilla caché en memoria que es el comportamiento por defecto.
- Gestión de los logs.
- Soporte de las notificaciones de eventos.
- Utilización de un Content Delivery Network a través de un mecanismo de middleware (por el momento, solo se soporta Amazon Cloudfront).
- Reporting automático de las métricas, que a día de hoy soportan el envío en productos de terceros New Relic o Bugsnag.

 El último punto de la lista es muy revelador de la filosofía de funcionamiento de las nuevas aplicaciones en Web y de lo que se llama *Web Oriented Architectures*: en lugar de incluir un sistema



de reporting, Docker prefiere apoyarse en servicios externos especializados en esta funcionalidad. Este enfoque acertado y desacoplado, permite obtener sistemas más evolucionados a lo largo del tiempo y participa de la urbanización de los sistemas de información. Se trata de una orientación extremadamente fuerte de las arquitecturas informáticas para los próximos años, ya implementada en varios dominios que han hecho el esfuerzo de normalización (agencias de viajes con OTA, de seguros con ACORD, de telefonía con e-TOM) y soportado hoy en día por nuevos dominios industriales.

Y todas estas funcionalidades solo son posibilidades de configuración del registro en sí mismo. Como hemos visto, también es necesario tener en cuenta las capacidades del servidor web que sirve de proxy incluso ampliando, soportando detrás del front varias versiones del registro, como se sugiere en la documentación Docker (<https://docs.docker.com/registry/deploying/#configure-nginx-with-a-v1-and-v2-registry>), de manera que se facilite la compatibilidad con los clientes Docker anteriores a la versión 1.6, primera en ser compatible con el API V2.

 Esta solución utiliza la tecnología Docker Compose para simplificar su implantación.

Para terminar, además de todo este aspecto de configuración, la implantación en producción de un registro también haría necesaria la supervisión de las máquinas, la gestión de la escalabilidad, etc.

En cualquier caso, montar y mantener un registro privado es una tarea pesada y conviene reflexionar antes de elegir esta vía, en lugar de pagar una cuenta privada en Docker Hub. Solo cuando se presentan necesidades de confidencialidad muy importantes, es justificable el sobre coste enorme del mantenimiento por sus propios medios de arquitectura como esta.

También se puede considerar un enfoque intermedio que consiste en separar el alojamiento y guardarlo bajo su control, dejando que usted realice la configuración compleja, justamente utilizando contenedores de proxy inverso Nginx para un registro seguro y ya preparados por otros. Por ejemplo, se pueden mencionar:

- <https://registry.hub.docker.com/u/marvambass/nginx-registry-proxy/>

- <https://registry.hub.docker.com/u/containersol/docker-registry-proxy/>

Por supuesto, es conveniente validar el nivel de confianza que se puede tener en estas imágenes. Las dos mencionadas anteriormente, son resultado de build automatizadas en Docker Hub y Dockerfile, así como los archivos de configuración, que son accesibles para validar el contenido, lo que es bueno.

En cualquier caso, son necesarias muy buenas razones para no utilizar los servicios cloud de registro Docker, porque el coste de mantenimiento por sus propios medios es extremadamente elevado, para llegar a un nivel inferior en términos de seguridad y robustez que el proporcionado a un precio menor por las alternativas en línea que vamos a mostrar ahora.

# Utilización de un servicio de registro en el cloud

## 1. Principio

No es una opción, si el trabajo necesario para mantener su propio registro le parece demasiado elevado frente a las ventajas, poniendo sus imágenes en línea en la parte pública de Docker Hub, por lo tanto sin ninguna seguridad respecto a quien las consume (y eventualmente se podría realizar ingeniería inversa en ella). Sin embargo, la utilización de un almacén privado donde el alojamiento de los recursos y del software se gestiona por un proveedor comercial, es sin lugar a dudas una solución mejor.

En este caso, el proveedor se encarga de:

- la gestión de los recursos físicos (servidores, equipamientos de red, climatización, etc.);
- el mantenimiento en condiciones operativas del sistema operativo;
- la elasticidad de los recursos en función de la petición, en el límite de las restricciones de consumo máximo fijadas por usted;
- la implantación de la aplicación de tipo registro, ya sea a través del registro ofrecido por Docker o por cualquier otro software que implemente correctamente el API del registro;
- proporcionar las métricas, logs, eventos de monitoring si desea analizarlas;
- proporcionar el ancho de banda necesario, así como el almacenamiento para que sus imágenes se guarden y difundan en buenas condiciones, también sometidas a contrato.


En este modo de funcionamiento, al usuario solo le queda cargar sus imágenes en el registro remoto y configurar las condiciones de acceso de dichas imágenes.

El panorama de las ofertas es muy amplio. Amazon ofrece EC2 Container Registry (<https://aws.amazon.com.sabidi.urv.cat/es/ecr/>), Google el producto Google Container Registry (<https://cloud.google.com/container-registry/>), Microsoft

el servicio Azure Container Registry (del que veremos una demostración más completa en una sección posterior), en resumen, todas las empresas importantes tienen su oferta, rivalizando en funcionalidades y rendimiento.

Hay empresas más pequeñas, incluso startups que se lanzan también en este mundo, destacando en uno u otro comportamiento particular como ventaja competitiva. De esta manera, Artifactory propone su registro integrado, Quay.io y Canister.io tienen ofertas que insisten más en la excelente integración con un proceso de despliegue continuo, etc.


Rancher ha tenido la excelente idea de ofrecer una comparativa de cuatro de estas ofertas en <http://rancher.com/comparing-four-hosted-docker-registries/>. Sea cual sea su elección, asegúrese principalmente de respetar el API V2 del registro Docker. Es el respeto estricto de este contrato técnico, el que le garantizará que pueda fácilmente pasar de un proveedor a otro.

 También puede ser una buena práctica no poner todos los huevos en la misma cesta y utilizar dos ofertas separadas, por ejemplo, con un registro para la difusión de las imágenes oficiales y otro destinado a sus desarrolladores y probadores. La ventaja es que en caso de error en su elección o de incidente permanente, hay una solución alternativa. Además, la comparación le puede permitir afinar su elección. Para terminar, la separación estricta por entornos diferentes, pueda ser un medio de garantizar la estanqueidad y una casi imposibilidad de error durante su proceso de puesta a disposición en staging.

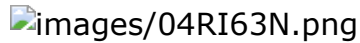
## 2. Oferta de pago de Docker Hub

Una posibilidad de las más sencillas para obtener este tipo de servicio, es pagar por una cuenta profesional en DockerHub, lo que le permitirá almacenar un determinado número de almacenes de manera privada (las cuentas gratuitas están limitadas a un único almacén privado).

A fecha de escritura de este libro, la oferta de Docker para la gestión de las imágenes comienza a nivel Enterprise Standard Edition (fuente: <https://www.docker.com/pricing>):

 images/04RI62N.png

Las tarifas asociadas son las siguientes:



La antigua oferta, más barata, que consistía solo en un número de almacenes privados, ya no se ofrece. Puede estar disponible todavía, pero el contacto con un vendedor le permitirá conocer las eventuales condiciones.

De manera general, la oferta de Docker tiene tendencia en los últimos años de convertirse en una oferta cada vez más integrada. Los buenos aspectos son la consolidación y la interoperabilidad. Los malos son una cierta rigidez de la oferta. Esta es la razón por la que aquí mostraremos otro enfoque.

### 3. Azure Container Registry


#### a. Preparación

Como se ha prometido más atrás, vamos a detallar el funcionamiento de la oferta de Microsoft en términos de registro privado disponible, como servicio cloud. La oferta de Microsoft se llama Azure Container Registry y, como su nombre indica, está disponible en el cloud de Azure. Este módulo de Azure permite disponer en algunos clics de su propio registro seguro en línea, con la mayoría de las funcionalidades imaginables y una ventaja adicional, que es el modo de facturación, donde solo se facturan el almacenamiento y el ancho de banda, en el modo llamada "clásico".

Se prevé otro modo más avanzado, en el momento de la escritura de la presente edición de este libro. Su principal aporte es una gestión más fina de las identidades y de los permisos, enlazando con el servicio Azure Active Directory. Además, estarán disponibles las webhooks para desencadenar acciones configurables durante los eventos en el registro.

La tarificación también cambiará respecto al modo clásico, con tres niveles de servicios, ilustrados a continuación (fuente: <https://azure.microsoft.com/es-es/pricing/details/container-registry/> a fecha 27/05/2018).



 Para establecer el ejemplo siguiente, será necesaria una cuenta Azur. Llegado el caso, puede obtener una cuenta de demostración gratuita con 200 \$ de crédito Azure en la

dirección <https://azure.microsoft.com/es-es/free/>. Una de las ventajas de esta oferta es que no se le podrá facturar siempre que no escoja explícitamente una oferta de pago (se utiliza el número de tarjeta de crédito para la validación de la identidad, pero si sobrepasa su crédito, no se le cargará automáticamente como hacen algunos proveedores; los servicios que haya implementado simplemente se detendrán).

## b. Creación del registro

Una vez constituida una cuenta, las operaciones son extremadamente sencillas:

Conéctese al portal Azure (<http://portal.azure.com>).

Con ayuda del botón **+** en la parte superior izquierda de la interfaz del portal, añada un recurso.


Filtre los recursos disponibles escribiendo `Azure Container Registry` en la caja de búsqueda:



Seleccione el resultado llamado **Azur Container Registry** (atención, el nombre pueda variar a lo largo del tiempo o dependiendo de la interfaz que utilice).

Pulse en el botón **Crear** en la parte inferior.

En la ventana que aparece, especifique un identificador para su registro. Se trata del código que se utilizará como sufijo `.azurecr.io` (`cr` para *container registry*) para componer la URL del registro.

 Este código debe ser único para todos los usuarios de la solución ofrecida por Azure. En caso contrario, la interfaz le avisará como se muestra en la siguiente imagen, a través de un icono en forma de punto de exclamación, a la derecha de la zona de texto y rodeado en rojo.



Proporcione el resto de información solicitada por la interfaz, como la suscripción a utilizar, la localización así como si desea crear o reutilizar un grupo de recursos. Si desea relacionar fuertemente el registro con un eventual conjunto Azure


Container Services, puede ser interesante ubicarlo en el mismo grupo de recursos.

La opción **Admin user** permite simplificar el acceso al registro, creando un administrador (con todos los permisos sobre el registro), que retoma el nombre utilizado más atrás para la parte variable de la URL de este. Actívela.

Pulse el botón **Crear** para validar las opciones elegidas.

Espere algunos segundos (incluso algunos minutos como máximo), durante la creación de los recursos solicitados y después acceda al grupo de recursos elegidos. En la siguiente captura de pantalla, el nombre elegido era **registry**.

images/77.png

 Una constatación es que se han creado solo dos recursos para establecer un registro de contenedores, a saber, el registro en sí mismo y una cuenta de almacenamiento. No es necesaria ninguna máquina virtual, lo que reduce el coste del conjunto, porque solo se calcula en función del ancho de banda consumido y del almacenamiento (este último siendo un recurso cloud muy barato).

Pulse en el recurso de tipo **Containers registry** en la lista. En nuestro caso, se trata de la línea titulada **gouigoux**.

images/78.png

Seleccione el menú **Access key** para encontrar las contraseñas necesarias para el acceso al registro. Como la opción **Admin user** está activada, el nombre de usuario será **gouigoux** y las contraseñas se generan aleatoriamente por Azure.

Se proporcionan dos contraseñas y es posible regenerarlas, lo que facilita el uso en caso de que estén comprometidas. Si el uso lo requiere, el administrador puede comunicar de esta manera una de las contraseñas y conservar la segunda para sus propios usos. De esta manera, la puesta a cero de la primera no impedirá sus usos internos de funcionar.

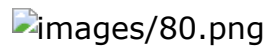
### c. Configuración


El registro creado dispone de numerosas funcionalidades profesionales como **Identity and Authorization Management** (gestión de las cuentas y de los permisos):



La asignación de acceso basada en roles permite hacerse cargo fácilmente de un número de usuarios elevado, sin que el coste de gestión aumente mucho.

Se basa en los siguientes roles:



 El autor piensa que es bueno recurrir a los tooltips de la interfaz para una descripción completa de los roles existentes. Aquí el del rol **Contributor** se muestra de manera informativa, en la captura de pantalla anterior.

La auditoría de actividad del registro también está integrada, con una interfaz de búsqueda de eventos en las trazas muy potente y que debería cubrir la gran mayoría de los usos, incluidos en producción industrial.





## Enfoques adicionales

### 1. El API del registro

Como curiosidad, todo registro que quiera estar conforme al modo de funcionamiento de Docker, debe implementar un API tal y como se especifica en <https://docs.docker.com/registry/spec/api/>.

De manera general, nos conectamos a un registro por el cliente Docker que transforme los comandos transmitidos en llamadas de API, para que nos ahorre complejidad. Pero para determinadas operaciones muy específicas (emisión masiva de imágenes, diagnóstico, etc.), pueda ser útil saber que este API existe y que expone todos los comandos utilizables por el cliente Docker, de manera estándar (REST/JSON).

Hemos mostrado un ejemplo muy sencillo cuando hemos validado el almacén de una imagen en un registro, accediendo a este último desde un navegador:

images/61.png

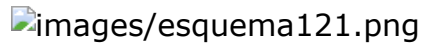
### 2. Implementación de un espejo

En las secciones anteriores del presente capítulo, se ha detallado abundantemente el funcionamiento de la caché local de imágenes. Hemos mostrado cómo las imágenes se almacenaban en una máquina host durante la compilación o mediante el uso de un comando `pull` que las trae para su utilización desde el registro Docker Hub u otro registro de la red, si es necesario. La gran utilidad de esta caché local se ha explicado de manera que si varias imágenes utilizan la misma imagen básica (`ubuntu` o `debian`, en nuestros ejemplos) en la misma máquina host, solo la primera puesta en marcha implicaría la descarga efectiva de estas imágenes desde el registro remoto Docker Hub. El resto de las ejecuciones siguientes de una imagen utilizando estas imágenes como base de un nivel cualquiera, serán mucho más rápidas.

Este mecanismo de caché es uno de los grandes progresos de Docker respecto a un sistema de virtualización, donde las máquinas completas se tenían que descargar, provocando

redundancias y un desperdicio enorme de ancho de banda. Solo, su implantación en grandes conjuntos de servidores, plantea problemas de eficiencia.

Imaginemos en efecto un cluster de máquinas hosts para alojar numerosos contenedores Docker:



Cada una de las máquinas del cluster va a tener el mismo comportamiento con una caché local, pero cada una deberá descargar en Internet la imagen básica durante la primera ejecución. Si se añade que en un cluster no es raro instalar máquinas desde cero tan pronto como se presenta un problema, entendemos que un nivel adicional de caché en la red local, sería una ventaja.

Afortunadamente es posible establecer un registro de tipo caché, que va a servir de espejo a Docker Hub (o a otro registro si es necesario). De esta manera, solo la primera descarga necesitará pasar por Internet, y el resto de máquinas utilizarán la caché sobre la red local (y por tanto, con un coste de ancho de banda reducido y mejores rendimientos), para enriquecer su propia caché local:



Para establecer esta infraestructura, vamos a utilizar de nuevo la imagen `registry`, pero declarándola como autónoma (es decir que no se trata de un registro de almacenamiento de referencia de las imágenes), y pasándole los argumentos adicionales. La documentación de Docker aconseja señalar `https://registry-1.docker.io` para las imágenes y `https://index.docker.io` para el índice.

```
docker run -e STANDALONE=false -e MIRROR_SOURCE=https://registry-1.docker.io -e MIRROR_SOURCE_INDEX=https://index.docker.io -p 5000:5000 registry
```

El resto es muy sencillo: es suficiente con pasar la dirección del contenedor expuesto de esta manera como valor de la opción de puesta en marcha `--registry-mirror` de Docker. Esta opción se puede pasar en la línea de comandos cuando lance el demonio Docker manualmente, lo que generalmente se reserva a las ejecuciones en modo depuración:

Su ubicación preferente es en el archivo `/etc/default/docker`,

```
docker --registry-mirror=http://IP_MAQUINA_ELEGIDA:5000 -d -D
```

en la variable DOCKER\_OPTS:

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development
testing).
#DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.
DOCKER_OPTS="--insecure-registry=miregistro:5000 -registry-
mirror=192.168.1.0.13:5000"

# If you need Docker to use an HTTP proxy, it can also be specified
here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files
go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

 No olvide volver a lanzar Docker para que se tengan en cuenta estas modificaciones.

Durante la implementación de un cluster a gran escala, esta buena práctica es casi obligatoria, en parte para no consumir demasiado ancho de banda, pero sobre todo, para asegurar un tiempo de puesta en marcha rápida de los contenedores en las nuevas máquinas que se integran en el cluster. Volveremos en un siguiente capítulo sobre la implementación efectiva de un cluster de máquinas hosts Docker.

## 3. Implementación de una caché para los paquetes

### a. Posible solución

Hay un tercer nivel al que se puede añadir una caché: se trata de la etapa de compilación de las `Dockerfile` en imágenes. Hemos visto más atrás que la compilación también utilizaba la estructura apilada de las imágenes para cachear al máximo las operaciones. También se ha mostrado las buenas prácticas de escritura y de programación de los `Dockerfile`, para favorecer el uso al máximo de la caché.

Una de las operaciones más largas durante la compilación de una imagen, es la instalación de los paquetes Linux necesarios.

Tomemos como ejemplo el siguiente Dockerfile, que hemos utilizado un poco más atrás:

```
FROM ubuntu:trusty
MAINTAINER jp.gouigoux@free.es

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask

COPY src/ .

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

El comando RUN instala Python, lo que lleva cierto tiempo (algunos minutos con un acceso ADSL correcto). Esta es la razón por la que este comando se ha ubicado lo más atrás posible en el archivo. El comando COPY que se ocupa de recuperar el código fuente, se sitúa debajo. En efecto, el código fuente cambia muy regularmente y es el motivo normalmente por el que existe una nueva versión que justifica volver a compilar una imagen. De esta manera, la instalación de Python se realiza una vez para todas y las nuevas versiones del código fuente no necesitan relanzar esta costosa operación.

Sin embargo, existen casos en los que es más complejo conservar este mecanismo propio. Esto puede venir del Dockerfile en sí mismo, así como de la máquina que sirve para la compilación. Por ejemplo, si estas máquinas son múltiples para acelerar el proceso de build de numerosas imágenes, la generación de la capa intermedia se hará varias veces.

Una buena práctica es utilizar el truco de imagen básica dedicada, mostrado un poco más atrás y utilizar la red para centralizarla, ya sea por un registro local o por un espejo como se ha mostrado anteriormente. Entonces, el Dockerfile de la imagen básica será:

```
FROM ubuntu:trusty
MAINTAINER jp.gouigoux@free.es

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask
```

```
ENTRYPOINT ["python"]
```

Y el de la imagen original se convertiría en:

```
FROM jpgouigoux:pythonbase
MAINTAINER jp.gouigoux@free.es

COPY src/ .

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

Para lo que es el uso de una caché para compartir la imagen básica entre las diferentes máquinas implicadas, no volveremos sobre esto que se ha mostrado justo antes.

## **b. Implementación de una caché de paquetes**

En los casos en los que los trucos anteriores no funcionen, por ejemplo en los entornos de desarrollo donde los requisitos previos cambian normalmente, otra solución es colocar un nivel de caché directamente a nivel de la descarga de los paquetes Linux. El software Aptitude utilizado en las distribuciones basadas en Debian, como Ubuntu, está preparada para este modo de funcionamiento y como sucede normalmente, lo más sencillo es establecer un contenedor Docker para implementar este modo de funcionamiento.

El Dockerfile que se debe utilizar es el siguiente:

```
FROM ubuntu:trusty
RUN apt-get update && apt-get install -y apt-cacher-ng
EXPOSE 3142
ENTRYPOINT ["/usr/sbin/apt-cacher-ng"]
CMD ["-c", "/etc/apt-cacher-ng", "Foreground=1"]
```

Las líneas se corresponden con las diferentes operaciones siguientes:

- FROM apunta a la imagen básica, como de costumbre.
- RUN instala la aplicación apt-cacher-ng, que es la aplicación elegida para realizar la caché de paquetes Aptitude (opción -y como es costumbre, para que el comportamiento no sea interactivo y la petición de confirmación de descarga se trate automáticamente como aceptada).
- EXPOSE publica el puerto 3142 como el puerto que se debe utilizar para acceder a la caché.

- ENTRYPOINT arranca el programa apt-cacher-ng cuando el contenedor se lanza.
- CMD permite especificar las opciones por defecto, a saber:
  - la utilización del directorio de configuración /etc/apt-cacher-ng (se trata de la ubicación por defecto, pero es útil precisarlo),
  - poner en primer plano el proceso. De lo contrario, la aplicación devolverá el control y el contenedor se detendrá inmediatamente después de haberlo lanzado, haciendo el conjunto no utilizable.

Entonces, el contenedor se puede arrancar justo después de la compilación de la imagen:


```
docker build -t cacher .
docker run -d -p 3142:3142 --name apt-cache cacher
```

La siguiente etapa consiste en asegurarse de que durante las compilaciones de imágenes Docker que llaman normalmente a los paquetes, sea esta caché la que se llame en lugar de los almacenes remotos. Para esto, es necesario añadir en la definición de estas imágenes una entrada en el directorio de configuración.

Imaginemos un Dockerfile que nos sirve de base para el desarrollo Python, y en la que normalmente vamos a añadir los paquetes a instalar. Podríamos crear una capa para cada nuevo paquete, pero también podemos volver a instalar todos los paquetes en la build, si incluimos la segunda línea del siguiente Dockerfile:

```
FROM ubuntu:trusty
RUN echo 'Acquire::http { Proxy "http://192.168.0.7:3142"; };'
>> /etc/apt/apt.conf.d/01proxy
RUN apt-get update && apt-get install -y build-essential python
python-dev
ENTRYPOINT ["python"]
```

El directorio de configuración de Aptitude se enriquece con una dirección de proxy que apunta a la caché que acabamos de establecer en un contenedor (no olvidar modificar la dirección IP para apuntar a la máquina correcta).

 De nuevo aquí, si desea no tener que entrar sistemáticamente esta operación en cada uno de sus Dockerfile con el riesgo

asociado de olvidarlo, se puede aplicar el truco de la imagen básica personalizada (ver más atrás).

Durante la primera compilación, el proceso lleva cierto tiempo, que podemos comprobar añadiendo el comando `time` delante del comando `docker build`:

images/64.png


Para asegurarse de no tener el efecto de la caché de Docker, sino comprobar únicamente el de la caché de los paquetes Aptitude, volveremos a lanzar el comando después de haber eliminado la imagen creada:

```
sudo docker rmi devpython
time sudo docker build -t devpython .
```

Los resultados muestran claramente la ganancia de tiempo, para este `Dockerfile` simplemente con tres paquetes:

images/65.png

Con un acceso ADSL casi estándar, la mejora es de 100 segundos sobre 135: la compilación necesita únicamente la cuarta parte del tiempo necesario en caso de que la caché de paquetes no se hubiera implementado. Un minuto y medio ganado en este sencillo ejemplo puede parecer poco, pero multiplicado por decenas de compilaciones cada día y por decenas de desarrolladores para una empresa de edición estándar, la ganancia de tiempo es sustancial al final del año, sobre todo acompañada de una economía de recursos importante (ancho de banda, principalmente). Para terminar, más allá de esta ganancia "técnica", lo más importante puede ser que una reducción de la espera para el desarrollador le pueda ayudar a permanecer concentrado en su tarea intelectual actual. Las consecuencias de no prestar atención a las fases de compilación, es un factor de errores en el desarrollo.

 Hemos mostrado un contenedor de caché de paquetes que, una vez detenido, va a perder su contenido almacenado, pero es posible (incluso recomendable) utilizar la gestión de los volúmenes para que se persista esta base de datos de caché, incluso en caso de reinicio del contenedor. Un comando `docker diff` [nombre del contenedor de caché de

paquetes] le muestra dónde se almacenan los archivos (a saber `/var/cache/apt-cacher-ng/`), información que también obtendría leyendo el archivo de configuración de la caché `/etc/apt-cacher-ng/acng.conf`.



## 5. IMPLEMENTACIÓN DE UNA ARQUITECTURA DE SOFTWARE

### Presentación de la aplicación ejemplo

Los anteriores capítulos se han basado en ejemplos que sobre todo, mostraban una utilización teórica de Docker. En los siguientes capítulos vamos a intentar hacer un recorrido un poco más industrial del uso de estos contenedores. La idea es mostrar esto de manera progresiva, parecido a lo que se ha hecho anteriormente con el registro privado, donde se ha partido de una instalación estándar y progresivamente se ha evolucionado hacia un registro casi listo para la producción.

#### 1. Arquitectura

En lo que sigue en este libro, nuestro hilo conductor será una aplicación compuesta de micro-servicios, en un enfoque SOA (*Service Oriented Architecture*, es decir arquitectura orientada a servicios). No se trata de una aplicación industrial real, porque la complejidad de esta haría que la lectura fuera más difícil y al final, empañaría los mensajes sobre Docker. Sin embargo, su arquitectura, así como la infraestructura que soporta su desarrollo, compilación e implantación, serán lo más parecido posible a las implantadas por el autor, en el marco de su profesión de director técnico de un constructor de software.

Nuestra aplicación ejemplo, llamada `mortgage`, sirve para la gestión de un préstamo (*mortgage* en inglés, de donde viene el nombre del proyecto). El caso de uso que queremos establecer, es el siguiente: "Como usuario identificado, quiero recibir un mail que contiene un informe sobre el mejor momento para reembolsar mi préstamo existente, aportando X € más un nuevo préstamo a un interés fijo de Y % como complemento".

El esquema siguiente muestra la arquitectura global de esta aplicación, que nos seguirá el resto de los capítulos:

images/esquema23.png

Las diferentes etapas utilizadas para construir el caso de uso explicado, son las siguientes:

- El portal web (`portal`) expone una interfaz que permite a un usuario crear una definición de un préstamo.
- Esta definición de préstamo se almacena en una base de datos MongoDB (`database`).
- El usuario pueda solicitar, para un directorio de préstamo dado, un informe sobre el mejor momento para emitir un reembolso parcial. El servicio de notificación (`notifier`) le enviará el informe por mail.
- Para esto, es necesario solicitar a un servicio (`reporting`) que construya este informe en PDF.
- Pero en sí mismo, necesita llamar a un servicio de cálculo de optimización del reembolso (`optimizer`).
- Para terminar, la optimización consistente en calcular el coste del reembolso para todas las fechas entre el día de simulación y el último día del préstamo, y después, determinar la fecha que minimice el coste. Para esto será necesario lanzar numerosos cálculos de plazos de préstamo, que se realizarán por varias instancias del servicio `calculator`.

El diagrama anterior indica las dependencias por las flechas. Observe que las flechas son inversas respecto a lo que parece lógico entre `optimizer` y `calculator`. Hemos utilizado una arquitectura de tipo "cola de mensajes", que permite a cada `calculator` consumir las tareas de cálculo y devolver el resultado, sin ser controlado por un nodo central de control, lo que es más eficaz desde el punto de vista del escalado: el punto central conserva de esta manera el menor trabajo posible, lo que aleja el momento en el que un aumento de la carga se convierta en un cuello de botella para el conjunto de la aplicación.


## 2. Instalación

El medio más sencillo de instalar el conjunto de este ejemplo, es utilizar la tecnología Docker Compose. Volveremos más adelante en detalle sobre los aspectos principales y diferentes opciones de esta aplicación, pero como lo necesitamos para este ejemplo, simplemente incluiremos las dos líneas de comandos necesarias para la instalación, tal y como se detalla en <https://docs.docker.com/compose/install/>, a saber:

```
curl -L
https://github.com/docker/compose/releases/download/1.16.1/docker-
compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

```
chmod +x /usr/local/bin/docker-compose
```

Una vez se ha instalado Docker Compose, vamos a poder lanzar la aplicación de ejemplo después de haber recuperado el código fuente en GitHub.

 El código fuente asociado a la aplicación ejemplo, está disponible en la cuenta GitHub Ediciones ENI, con el nombre de proyecto `mortgage`. Para que la aplicación pueda continuar evolucionando, conservando la pertinencia de los extractos que se dan en este libro, se ha fijado una versión 1.0 para que se corresponda con la primera edición del presente libro, y después una segunda versión 2.0 para otro libro sobre Docker también en Ediciones ENI y para terminar, una tercera versión 3.0 correspondiente a la segunda edición que tiene entre manos. Encontrará el código de los extractos mostrados a continuación, desde el enlace <https://github.com/EditionsENI/mortgage/tree/v3.0>.

Recupere el código fuente del ejemplo con el siguiente comando:

```
git clone https://github.com/EditionsENI/mortgage
```

Posiciónese en el directorio `mortgage`.

Sitúese en la versión 3.0, que se corresponde con la utilizada para el presente libro:

```
git checkout v3.0
```

Construya todas las imágenes necesarias para la aplicación, utilizando el siguiente comando (volveremos más adelante sobre su funcionamiento):

```
docker-compose build
```

El siguiente comando indica a Docker Compose que lance tres instancias del servicio `calculator`:


```
docker-compose scale calc=3
```

Los únicos argumentos que necesita la aplicación, son el identificador y la contraseña de una cuenta Gmail, para poder enviar mensajes.

Proporcione su información de cuenta asignando valores

correctos a las variables de entorno llamadas como sigue:


```
export SMTP_AUTH_LOGIN=su.identificador@gmail.com
export SMTP_AUTH_PASSWORD=su.contraseña
```

 Para que el envío de mails funcione, es necesario activar la autorización sobre <https://www-google-com.sabidi.urv.cat/settings/security/lesssecureapps>. Volveremos un poco más adelante sobre este punto.

Lance el conjunto de micro-servicios una única vez (esto hace que se arranque la aplicación ejemplo), utilizando el siguiente comando:

```
docker-compose up
```

Ahora la aplicación está disponible en <http://localhost:8080>. Después de algunos segundos de inicialización, durante los cuales debería ver los logs de puesta en marcha en la consola, aparece una visualización parecida a la siguiente:

 images/05RI02.png

### 3. Utilización

Cuando arranca la aplicación, el usuario ve la siguiente pantalla en su navegador web (si, como el autor, usted despliega la aplicación en una máquina con una dirección DNS válida, utilizará este nombre en lugar de localhost):

 images/05RI20N.png

La primera acción consiste en crear un préstamo, rellenando las casillas según las instrucciones incluidas y después, haciendo clic en el botón **Create**. Por ejemplo:


 images/05RI21N.png

Esto va a crear un préstamo empezando en febrero del 2001 (decimotercer mes desde el 01/01/2000), con un capital (**Amount borrowed**) de 100.000 € y una duración (**Repayment length**) de 20 años (240 meses), a un interés fijo (**Rate if fixed**) del 3,5 % (preste atención para utilizar correctamente el separador decimal correspondiente al punto, la aplicación ejemplo no soporta los argumentos de regionalización). Cuando se utiliza el botón de

creación (**Create**), el software transfiere automáticamente la referencia del préstamo creado al segundo formulario, que está dedicado a la optimización del reembolso del préstamo. Esta vez, las variables a indicar son la cantidad del reembolso fijo (**Amount repaid**), por ejemplo 20.000 € y el interés fijo de sustitución (**Replacement fixed rate**), por ejemplo 2,5 %. El usuario también debe indicar su dirección de mail para recibir el informe generado para él.



Después de algunos segundos, incluso minutos de cálculo, se obtiene el informe por mail.

 Para que el ejemplo sea funcional, conviene crear un préstamo que todavía esté en curso de reembolso durante la emisión de una petición de informe de optimización de reembolso. De esta manera, si lanza este ejemplo después de octubre del 2017, será necesario aumentar la duración o bien cambiar el mes de inicio a un valor superior a 13.

## 4. Herramienta

En este capítulo dedicado a la implantación práctica de Docker, nos vamos a interesar particularmente por la manera en la que se compilan los diferentes servicios y en qué nos puede ayudar Docker respecto a la gestión del ciclo de vida de la aplicación, hasta el despliegue. La arquitectura propuesta mezcla el diseño de numerosos frameworks de desarrollo, para que cada lector pueda encontrar fácilmente un ejemplo correspondiente a su flujo habitual de trabajo.

Además, el hecho de establecer una arquitectura fuertemente desacoplada con varios servicios, será útil para demostrar las relaciones posibles entre contenedores, así como la manera de desarrollar varios contenedores a la vez, gracias a Docker Compose, que solo habíamos introducido sin explicarlo por el momento.

## 5. Aspectos principales del ejercicio

### a. Una palabra sobre las arquitecturas de micro-servicios

Docker se utilizaba mucho en arquitecturas de micro-servicios. Parecía lógico por tanto utilizar un ejemplo que, aunque de manera muy simplificada siguiera esta arquitectura, donde cada responsabilidad funcional se confía a una aplicación autónoma que expone un API desacoplado del resto.

Sin entrar en detalles, las arquitecturas de micro-servicios son modelos de funcionamiento de aplicaciones web en las que todas las responsabilidades se han desacoplado de manera muy fina, de manera que cada servicio (que es independiente del resto, en el sentido de que tiene su propio proceso y su ciclo de vida dedicado), solo realiza una única funcionalidad de negocio. Esta es la razón por la que un escenario tan sencillo como enviar un informe con la mejor fecha de reembolso, necesita al final seis servicios de software diferentes.

En un primer momento, este modo de funcionamiento pueda parecer muy complicado respecto a la creación de una única aplicación monolítica, pero su valor reside - entre otras ventajas - en el hecho que de una arquitectura como esta, se mantiene de manera muy sencilla. Si el desacople entre servicios está bien pensado, la modificación de un servicio se hace con una garantía de limitación de impacto sobre el resto. Además, se facilita la mutualización, gracias a que el componente de software correspondiente es realmente autónomo. Para terminar, el hecho de que la granularidad sea muy fina, permite gestionar una escalabilidad con un equilibrado muy concreto entre los diferentes servicios.

Vamos a demostrar una por una todas estas ventajas, de las que Docker participa de manera destacada.

## **b. Relación con la programación SOLID**

De una determinada manera, el desacoplamiento de los micro-servicios se parece al que se realizaría entre las clases si la aplicación estuviera codificada de manera antigua, con todas las funciones en un mismo proceso. La separación de las responsabilidades se haría según los principios SOLID de POO (programación orientada a objetos).

También es elocuente que se apliquen los cinco principios del método SOLID, tanto a la definición de micro-servicios como a la definición de las clases en POO:

- **S (Single Responsibility Principle):** el principio de responsabilidad único se corresponde con el hecho de que un servicio solo debe realizar una única funcionalidad.

- **O (Open Closed Principle):** puede ser el principio que mejor cumple Docker, que permite cambiar fácilmente un contenedor por otro que responda a la misma interfaz (abierto al cambio), garantizando una gran estabilidad y estanqueidad por el sistema de capas en modo solo lectura (cerrado a la modificación).
- **L (Liskov Substitution Principle):** un servicio se debe poder sustituir por una implementación más reciente, soportando la interfaz contractual y continuar soportando todos los casos de uso existentes, con el mismo resultado. En la práctica, este principio se asegura por las pruebas del tipo de las realizadas por el software SOAPUI.
- **I (Interface Segregation Principle):** una interfaz web por servicio. Cada servicio solo debe tener una única razón para cambiar.
- **D (Dependency Inversion Principle):** la inversión de dependencias es justamente sobre lo que hablaremos, con el enlace inverso entre los dos servicios `.NET optimizer` y `calculator`. En lugar de acoplarlos fuertemente llamando a las calculadoras para el optimizador (lo que además le dará la responsabilidad de equilibrar sus tareas), partimos del principio de que un contrato le permite saber cómo van a intercambiar la información y después, hacer que la relación sea inversa, a saber, que cada calculador sea responsable de pedir un trabajo al optimizador, enviándole los resultados correspondientes.

# Creación de la arquitectura de ejemplo

## 1. Principios de construcción

El resto del presente capítulo se dedica a describir la recreación de esta aplicación ejemplo, basándose exclusivamente en un editor de texto y Docker. Por lo tanto, puede eliminar el directorio `mortgage` que se había clonado desde GitHub y volver a crear la arborescencia vacía siguiente:

```
mortgage/  
├── calculator  
├── notifier  
├── optimizer  
├── portal  
└── reporting
```

En las secciones siguientes vamos a recrear uno a uno los servicios correspondientes a los diferentes directorios y después, crearemos el archivo que permite a Docker Compose controlar el despliegue del conjunto.

La idea detrás de este enfoque es mostrar una implantación práctica de Docker. En particular, el hecho de que utilicemos numerosas plataformas y lenguajes diferentes, es algo voluntario porque cada una necesita ciertos aspectos sutiles de funcionamiento o configuración respecto a Docker. El objetivo no es hablar de estas plataformas de desarrollo para explicar su funcionamiento, sino mostrar el mínimo necesario que hay que entender para ver cómo funcionan en un contenedor Docker.

Tanto como sea posible, solo mostraremos pequeñas porciones de código propiamente dichas, cuando estén relacionadas con la explicación del contenido de `Dockerfile`, que será el principal archivo de estudio. Para todos los archivos anexos que no presentan interés respecto a Docker, dejaremos al lector que vuelva a copiar las porciones de código correspondientes desde el proyecto Git Hub, porque su interés es limitado respecto a nuestro tema central.

El orden elegido para la explicación detallada de los servicios, es el inverso al de las referencias, de manera que en cada etapa, podamos fácilmente probar lo que se ha implementado.



## 2. Detalles del servicio optimizer


Por lo tanto, la primera de las imágenes a establecer es el servicio de optimización, que es el núcleo de la aplicación y que no tiene dependencia técnica respecto al resto de servicios, incluso si hay una dependencia funcional respecto al servicio de cálculo de los préstamos.

### a. Funcionamiento

La plataforma utilizada para el servicio de optimización es ASP.NET Core 2.0. Como Microsoft proporciona una imagen básica para esta tecnología, podemos directamente basarnos en ella para el archivo `Dockerfile` de la presente imagen.

En un primer momento, se va a escribir el contenido aplicativo. Para esto, vamos a crear la siguiente arborescencia de archivos y directorios:

```
optimizer/  
├── Dockerfile  
└── src/  
    ├── Job.cs  
    ├── JobsController.cs  
    ├── MortgageController.cs  
    ├── Mortgage.cs  
    ├── Program.cs  
    ├── Startup.cs  
    ├── optimizer.csproj  
    └── wwwroot/
```

 Observe que el directorio `wwwroot` está vacío, pero obligatoriamente debe estar presente para el proyecto ASP.NET Core 2.0 que vamos a definir. En el caso contrario, un error indicará que es necesario.

El archivo `optimizer.csproj` sirve de definición del proyecto, de manera que se expongan los argumentos imprescindibles para un ejecutable. En este caso, esto consiste principalmente en especificar la versión del framework destino, algunas dependencias y la inclusión del directorio `wwwroot` en el proyecto:

```
<Project Sdk="Microsoft.NET.Sdk.Web">  
  
  <PropertyGroup>  
    <TargetFramework>netcoreapp2.0</TargetFramework>  
  </PropertyGroup>
```

```

    <ItemGroup>
      <Folder Include="wwwroot\" />
    </ItemGroup>

    <ItemGroup>
      <PackageReference Include="Microsoft.AspNetCore.All"
Version="2.0.0" />
    </ItemGroup>

    <ItemGroup>
      <DotNetCliToolReference Include="Microsoft.VisualStudio.
Web.CodeGeneration.Tools" Version="2.0.0" />
    </ItemGroup>

  </Project>

```

Respecto al código fuente, mostraremos algunas porciones necesarias para entender las pruebas del servicio. El archivo `Mortgage.cs` define la estructura de un préstamo:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Optimizer
{
    public class Mortgage
    {
        public int startMonth { get; set; }
        public int repaymentLengthInMonths { get; set; }
        public decimal amountBorrowed { get; set; }
        public bool isRateFixed { get; set; }
        public decimal fixRate { get; set; }
    }
}

```

El archivo `Job.cs` define una unidad de operación que se ejecutará por uno de los servicios de tipo `calculator`:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Optimizer
{
    public class Job
    {
        public string Id { get; set; }
        public int RepaymentMonth { get; set; }
        public decimal RepaymentAmount { get; set; }
        public decimal ReplacementRate { get; set; }
        public Mortgage MortgageDefinition { get; set; }
        public bool Taken { get; set; }
        public bool Done { get; set; }
        public decimal TotalMortgageCost { get; set; }
    }
}

```

```
}  
}
```

Más adelante mostraremos cómo el calculador ejecuta este trabajo. Por el momento, es suficiente saber que el servicio `optimizer` crea jobs, deja a los calculadores resolverlo y cuando se tratan (buleano `Done` pasa a `true`), recupera el resultado para comparar las diferentes posibilidades de reembolso y determinar lo que cuesta más barato, dejando la propiedad `TotalMortgageCost`, que da el coste total del préstamo en función de las condiciones de reembolso (este es el valor, complejo de calcular, que ofrecen las instancias del servicio `calculator`).

El archivo `Startup.cs` solo arranca MVC 6, de manera que los dos controladores puedan exponer los API necesarios para el optimizador:

- El controlador de jobs expone un API REST sencillo que permite leer los jobs y modificarlos. Este es el API que usarán las calculadoras para mirar si quedan jobs por ejecutar, indicar cuándo se carga el job y devolver el carácter "terminado" al mismo tiempo que el resultado.
- El controlador de préstamo expone un sencillo servicio `BestRepaymentDate`, que devuelve la mejor fecha de reembolso en función del capital puesto a disposición para el reembolso y del nuevo interés. Para implementar este servicio, se crean tantos jobs de cálculo como meses posibles para el reembolso (entre el mes actual y el último mes de reembolso), pasa esta lista al primer controlador, y después simplemente espera (con un plazo máximo) a que todos los jobs se hayan tratado. Después, recorre los resultados para devolver el mejor.

Al contrario que con las versiones preliminares de ASP.NET Core, es necesario acompañar el archivo `Startup.cs` de un archivo adicional, llamado en nuestro caso `Program.cs`, y que contiene una función `Main` que va a controlar la construcción y la carga de la clase `Startup`:

```
using System;  
using System.Collections.Generic;  
using System.IO;  
using System.Linq;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore;  
using Microsoft.AspNetCore.Hosting;  
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.Logging;
```

```

namespace Optimizer
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

La definición de una API REST en ASP.NET Core 2.0 es muy sencilla:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace Optimizer
{
    [Route("api/[controller]")]
    public class JobsController: Controller
    {
        static internal object Bloqueo = new object();
        static internal List<Job> Jobs = new List<Job>();

        [HttpGet]
        public IEnumerable<Job> GetAll()
        {
            return Jobs;
        }

        [HttpPut("{id}")]
        public void Put(string Id, [FromBody] Job value)
        {
            lock (Bloqueo)
            {
                Jobs.RemoveAll(j => j.Id == Id);
                Jobs.Add(value);
            }
        }
    }
}

```

Todos los archivos fuente se deben ubicar en el directorio `src`. A continuación, podemos pasar al más importante, es decir, el archivo `Dockerfile`.

 En una aplicación de nivel industrial, no se utilizaran variables estáticas y la gestión de la concurrencia no se realizaría con un

mecanismo tan sencillo. Este enfoque simplista se utiliza para no complicar el programa, cuyo objetivo es servir de soporte a una implantación de un contenedor Docker.

## **b. Integración en Docker**

El contenido de Dockerfile es el siguiente:

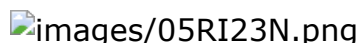
```
FROM microsoft/dotnet:2.0-sdk AS builder
WORKDIR /app
ADD /src/optimizer.csproj /app
RUN dotnet restore
ADD /src /app
RUN dotnet publish -o output
FROM microsoft/aspnetcore:2.0
COPY --from=builder /app/output /app
WORKDIR /app
```

Es conveniente comenzar explicando una novedad de las últimas versiones de Docker: Dockerfile multi-imágenes. Tradicionalmente, un archivo Dockerfile solo contiene una única instrucción FROM que describe la imagen que servirá de base para la imagen a generar por la compilación deducida del Dockerfile, con ayuda del comando `docker build`. Sin embargo, en muchos casos, el contenido creado por Dockerfile necesita actividades de compilación y la presencia de numerosas herramientas que, una vez que se haya creado la imagen, ya no son necesarias para su ejecución y ocupan espacio para nada.

La solución propuesta por Docker es muy elegante, porque permite fundirse en una imagen básica para ejecutar una parte de los comandos y después, bascular sobre otra (normalmente más ligera), para terminar las operaciones de compilación. Entre las dos, una opción permite recuperar los resultados de las primeras etapas para que la segunda se beneficie.

En nuestro caso, la primera etapa de preparación de la imagen se basará en la imagen `microsoft/dotnet:2.0-sdk`, que se corresponde con el Software Development Kit para .NET Core 2.0 y por tanto, contiene todas las herramientas necesarias para la compilación de aplicaciones en este lenguaje. Se trata de aplicaciones en forma de consola o aplicaciones web basadas en ASP.NET. La segunda etapa utilizará la imagen `microsoft/aspnetcore:2.0`, que contiene el runtime necesario para la ejecución de ASP.NET Core 2.0 y ninguna otra.

Como se pueda ver a continuación, la diferencia es el tamaño:



La segunda etapa de `Dockerfile` consiste en colocar el directorio de trabajo en `/app`, como recomienda la documentación de la imagen básica ofrecida por Microsoft. Este directorio es, por convención, el que contendrá el contenido de la aplicación .NET.

A continuación, la operación de preparación del programa se va a realizar en dos etapas, para aprovechar al máximo el mecanismo de caché en la generación de una imagen Docker. En una primera etapa, se copia el archivo que define el proyecto ASP.NET, llamado `optimizer.csproj` y se lanza el comando `dotnet restore`, que se base en él. Esto va a provocar la descarga de todas las dependencias necesarias para el proyecto. Es en una segunda etapa cuando el código fuente en sí mismo se va a copiar (segunda operación `ADD`) y la compilación se lanzará con el comando `dotnet publish`. La ventaja de esta manera de proceder es que, como las referencias cambian en general menos habitualmente que el código, las dos primeras operaciones tendrán las máximas oportunidades de utilizar la caché. El comando `dotnet restore` es claramente una operación costosa, porque necesita descargas (operaciones más largas que una sencilla compilación local de código fuente). Cuando se modifica el código fuente, se volverá a lanzar la única operación `dotnet publish` y las dependencias se recuperan de la caché local, lo que mejora mucho la generación. Este tipo de optimización puede parecer anodina cuando se considera de manera unitaria, pero en un contexto de edición de software o decenas de desarrolladores en un editor de software, realizan modificaciones centenares, incluso millares de veces cada día, hacerlo de otra manera sería un enorme lío de recursos, así como una pérdida de tiempo difícilmente justificable.

Después de esta primera etapa que consistía en crear el software contenido en la imagen, llegamos a la segunda orden `FROM`, que marca el paso a la segunda etapa de la generación. La imagen `microsoft/dotnet:2.0-sdk` contenía todas las herramientas necesarias para producir la aplicación compilada, pero para su ejecución, la imagen `microsoft:aspnetcore:2.0` es suficiente. La segunda orden `FROM` bascula esta imagen básica. Por supuesto, los resultados de la primera parte no se pierden y sigue la orden `COPY` o más concretamente, la opción `--from`,

que va a permitir encontrarlas. Esta operación copia los archivos, pero en lugar de dejarlos en el contexto, los toma en la salida de la etapa identificada por la palabra clave proporcionada. En este caso, la palabra `builder` hace referencia al alias proporcionado durante el primer `FROM` del archivo `Dockerfile`, detrás de la palabra clave `AS`. De esta manera, la salida de la operación de compilación, que se había enviado al directorio `output`, se copiará en el directorio `/app`, que se convierte en el directorio actual usando el siguiente comando `WORKDIR`.

Para terminar, el comando `ENTRYPOINT` completa la definición de `Dockerfile`, especificando el comando a lanzar para que ASP.NET arranque en el entregable creado anteriormente, y que consiste simplemente en lanzar el comando `dotnet` con el nombre del archivo a lanzar como argumento, que por defecto es el nombre del proyecto seguido de la extensión `.dll`.

Un ojo experto notará que no se ha utilizado ninguna palabra clave `EXPOSE` en el archivo `Dockerfile` generado. ASP.NET va a escuchar por defecto en el puerto 80, y Docker expondrá esto directamente.

Nuestro `Dockerfile` ya está preparado para ser compilado como una imagen llamada `optimizer`, posicionándose en el directorio con el mismo nombre:

```
docker build -t optimizer .
```

En caso de error de creación de la imagen, lo más sencillo será comparar los archivos con los de la cuenta GitHub de Ediciones ENI, en el proyecto `mortgage`.

### c. Pruebas

Para probar el servicio, lanzamos un contenedor con un comando como sigue:

```
docker run -d -p 5004:80 --name testoptim optimizer
```

Podemos constatar inmediatamente en las trazas, que el servicio se ha arrancado correctamente:

```
jpg@Ubuntu1410:~/mortgage/optimizer$ docker logs testoptim
Started
```

Una sencilla llamada desde un navegador es insuficiente para desencadenar un cálculo. Para esto, es necesario utilizar una herramienta como Postman (disponible en Chrome) o RESTClient (para Firefox). Se pueden utilizar valores de prueba como los mostrados más atrás, para comprobar que la llamada del servicio web crea bien los jobs de cálculo. Por ejemplo, la URL a llamar sería la siguiente:

```
http://localhost:5004/api/Mortgage/BestRepaymentDate?AmountRepaid=20000&ReplacementFixRate=0.025
```

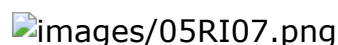
El contenido del cuerpo de la petición (que se debe enviar en modo POST), sería el contenido JSON siguiente:

```
{
  "startMonth": 13,
  "repaymentLengthInMonths": 200,
  "amountBorrowed": 20000,
  "isRateFixed": true,
  "fixRate": 0.035
}
```

Para terminar, se añadirán los encabezados (*headers* en inglés) siguientes:

```
Accept: text/plain
Content-Type: application/json
```

En una herramienta como Postman, el resultado que se obtendría después de un minuto de espera, sería una fecha nula, como se observa a continuación.





Esto se explica porque se trata del comportamiento por defecto del servicio si no llega a lanzar los cálculos. Una vez que se realice esta llamada, un vistazo a los logs permite comprobar que el servicio ha realizado bien su parte del trabajo esperado:

```
jpg@Ubuntu1410:~/mortgage/optimizer$ docker logs testoptim
Started
29 jobs created
```

Dando por hecho que no hemos conectado calculadores por el momento, el servicio `optimizer` crea los jobs de cálculo, pero estos no se tienen en cuenta. Por consiguiente, el servicio espera un minuto y después termina sin resultado. Pero, como se ha explicado un poco más atrás, se trata del comportamiento esperado y las pruebas se pueden considerar positivas.



 Para validar la presencia de los jobs, es posible llamar a `http://localhost:5004/api/Jobs`, que muestra la lista JSON de los jobs creados, como se puede ver en la siguiente captura de pantalla.

 images/05RI08.png

## d. Observaciones

Hasta ahora, los contenedores que necesitaban un puerto recibían un valor fijo en el comando, pero este tipo de prácticas no mantiene la carga. Desde el momento en que se trata de lanzar varios contenedores, en lugar de crear un plan explícito de asignación de puertos y correr el riesgo de equivocarse, es más sencillo dejar que Docker elija un puerto y se adapte. Esto es lo que vamos a hacer a continuación, lanzando los servicios como sigue:

```
docker run -d -P --name testoptim optimizer  
docker ps
```

El primer comando permite exponer los puertos especificados en `Dockerfile`, pero sin especificar puerto público correspondiente. El segundo permite encontrar el puerto automáticamente asociado por Docker y este es el puerto público, el que hubiéramos utilizado posteriormente para probar el servicio.

## 3. Detalles del servicio de calculador

### a. Funcionalmente

La siguiente etapa lógica consiste en establecer un servicio de cálculo, cuyas misiones serán:

- encargarse de los jobs,
- resolverlos calculando el coste del préstamo,
- indicar que el resultado se ha enviado al servicio `optimizer`.

La razón por la que preferimos que `optimizer` no se encargue de controlar los servicios de cálculo, sino que sea un oyente pasivo de lo que realizan estos últimos, es que esto favorece la

escalabilidad porque el único servicio que sigue siendo central no tiene casi nada que hacer. Si el servicio que hace de cuello de botella tiene muchos recursos que gestionar, llega más rápido el momento en el que añadir servicios secundarios ya no sirve de nada. En el peor de los casos, esto también puede ralentizar el conjunto, dando todavía más trabajo al servicio central, que ya trabaja al máximo de sus capacidades.

Por lo tanto, es el servicio `calculator` el que se va a encargar de listar los jobs, elegir cual calcular, resolver el coste total y reportar el resultado a la lista expuesta por `optimizer`. En cada etapa, también se indicará el número de jobs que quedan por tratar.

No entramos en los detalles del código fuente, que solo tienen un único archivo `Program.cs` (se trata de una aplicación en modo consola, basada en .NET Core 2.0, y que utilizará la misma imagen básica que el anterior servicio y otra imagen Microsoft, para la segunda etapa un poco más genérica, porque se adapta menos que la anterior en el caso específico de ASP.NET), pero retomamos algunos puntos que tienen importancia respecto a la arquitectura ejemplo y a su integración en Docker.

En primer lugar, la aplicación no era un servidor web en el sentido habitual, sino una aplicación en modo consola, que escucha por un puerto. La ejecución en el `Dockerfile` será necesariamente diferente. Además, no habrá que exponer un puerto. Por el contrario, será necesario prever una variable para especificar a la aplicación qué URL le da acceso a la lista de jobs. Este es el trabajo realizado por el fragmento de código siguiente:

```
public class Program
{
    public void Main(string[] args)
    {
        var config = new Configuration().AddEnvironmentVariables();

        string urlOptimizer = Environment.
GetEnvironmentVariable("URLOptimizerJobs") ??
"http://optimizer/api/Jobs/";
        Console.WriteLine("Mortgage calculation service listening
to {0}", urlOptimizer);
    }
}
```

La función `Main` arranca el programa almacenado en la clase `Program`. Lo primero que hace es leer la configuración de ejecución y en particular, el contenido de la variable de entorno `URLOptimizerJobs`. El valor por defecto es `http://optimizer/api/Jobs`, por una buena y sencilla razón: Docker accede por defecto al resto de servicios de un

archivo Docker Compose por medio de su nombre de servicio (volveremos sobre este punto un poco más adelante). Además, una traza permite comprobar el valor utilizado por esta URL.

El programa está construido sobre un bucle constante y llama a la lista de los jobs para ver si hay alguno que todavía no se haya tenido en cuenta (propiedad booleana Taken):

```
var httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Accept.Clear();
httpClient.DefaultRequestHeaders.Accept.Add(new
    MediaTypeWithQualityHeaderValue("application/json"));
List<Job> jobs = httpClient.GetAsync(urlOptimizer).Result.Content.
    ReadAsAsync<List<Job>>().Result;
List<Job> remainingJobs = jobs.FindAll(j => !j.Taken);
```

Si no hay, el programa marca una pausa y después vuelve a preguntar. La pausa es necesaria para no agotar los recursos. Por el contrario, si se acaba de crear un job y todavía no se ha tenido en cuenta por otro calculador, el servicio se lo va a asignar pasando su valor Taken a true y después, retomando el cálculo propiamente dicho, que aquí se simula por una sencilla espera de 20 milisegundos.

```
httpClient.PutAsJsonAsync<Job>(urlOptimizer + Taken.Id,
    Taken).Result.EnsureSuccessStatusCode();
Task.Delay(20).Wait();
```

No hay ninguna ventaja en complicar la comprensión o el funcionamiento del programa en su conjunto, haciendo un cálculo real del interés de préstamo. El programa evolucionará en la cuenta GitHub asociada. La versión 3.0 es la que se corresponde con los extractos en este libro, pero en versiones futuras puede que aparezcan cálculos más complejos y realistas, en función de las necesidades para otros proyectos.

## b. Integración en Docker

Un vistazo al archivo `calculator.csproj` revela que contiene información interesante para la integración del software:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNet.WebApi.Client"
      Version="5.2.3" />
  </ItemGroup>
</Project>
```

```

    <PackageReference Include="System" Version="4.1.0311.2" />
    <PackageReference Include="System.Collections"
Version="4.3.0" />
    <PackageReference Include="System.Configuration"
Version="2.0.5" />
    <PackageReference Include="System.Console" Version="4.3.0" />
    <PackageReference Include="System.Linq" Version="4.3.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.3" />
    <PackageReference Include="System.Runtime.Serialization.Json"
Version="4.3.0" />
    <PackageReference Include="System.Threading" Version="4.3.0" />
    <PackageReference Include="Newtonsoft.Json" Version="6.0.1" />
    <PackageReference Include="System.Security.Permissions"
Version="4.0.0" />
  </ItemGroup>
</Project>

```

En primer lugar, la lista de las dependencias hace que aparezcan las librerías que se utilizarán por nuestro software, en particular las utilizadas por el cliente HTTP. Algunas dependencias se hacen necesarias por otras, como `NewtonSoft.Json` que necesita a `System.Security.Permissions`. En el momento en el que se escriben estas páginas, se utilizan las versiones más recientes, pero el lector encontrará que han evolucionado cuando tenga este libro entre las manos.

Otra observación es que esta vez que, el tipo de salida es `Exe`. Esta es la indicación para `.NET` de que se trata de una aplicación de tipo consola.

El `Dockerfile` correspondiente a este servicio, se presenta como sigue:

```

FROM microsoft/dotnet:2.0-sdk AS builder
WORKDIR /app
ADD /src/calculator.csproj /app
RUN dotnet restore
ADD /src /app
RUN dotnet publish -o output

FROM microsoft/dotnet:2.0-runtime
COPY --from=builder /app/output /app
WORKDIR /app
ENV URLOptimizerJobs=http://optimizer/api/Jobs/
ENTRYPOINT ["dotnet", "calculator.dll"]

```

La gran mayoría del archivo no cambia mucho respecto al del servicio `optimizer`, lo que es lógico porque utilizan la misma tecnología. El mecanismo de dos etapas ya se ha explicado y no volveremos sobre él.

Por el contrario, la variable de entorno prefijada por el comando `ENV` permite pasar un valor, para apuntar el calculador sobre el punto que queremos. En nuestra sencilla utilización del ejemplo

de aplicación, no tendremos ocasión de cambiar su valor, porque el servicio utilizado para listar los jobs siempre será el lanzado por el mecanismo Docker Compose.

Otro cambio es el de la imagen `FROM` utilizada para la segunda etapa de generación de la imagen. En lugar de `microsoft/aspnetcore:2.0`, se utiliza `microsoft/dotnet:2.0-runtime`. Como indica su etiqueta, se trata de una imagen que solo contiene las herramientas necesarias para la ejecución (`runtime`) del software compilado en la primera etapa gracias a las herramientas (`sdk`), por lo que no tenemos necesidad posteriormente.

### c. Pruebas

No mostraremos de nuevo el comando de build de la imagen, porque consideramos que se conoce perfectamente. Solo es importante el nombre para que el ejemplo sea más fácil de seguir; le recomendamos utilizar `calculator`.


La etapa más interesante es la que se corresponde con `dotnet restore`, que descarga las librerías .NET Core necesarias para la correcta compilación (y después ejecución) del proceso:

```
Step 4/10: RUN dotnet restore
---> Running in ade77c3ladfb
Restoring packages for /app/calculator.csproj...
Installing runtime.fedora.23-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.rhel.7-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.ubuntu.16.04-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.opensuse.42.1-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.ubuntu.14.04-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.fedora.24-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.ubuntu.16.10-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.debian.8-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.osx.10.10-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing runtime.opensuse.13.2-x64.runtime.native.System.
Security.Cryptography.OpenSsl 4.3.2.
Installing System.Xml.XmlSerializer 4.3.0.
Installing runtime.native.System.Security.Cryptography.
OpenSsl 4.3.2.
Installing System.Private.DataContractSerialization 4.3.0.
Installing System.Net.Http 4.3.3.
Installing System.Runtime.Serialization.Json 4.3.0.
```

```
Installing System 4.1.0311.2.
Installing System.Security.Permissions 4.4.0.
Installing Newtonsoft.Json 6.0.1.
Installing System.Configuration 2.0.5.
Installing Microsoft.AspNet.WebApi.Client 5.2.3.
/app/calculator.csproj: warning NU1603: calculator depends
on System.Security.Permissions (>= 4.0.0) object System.Security.
Permissions 4.0.0 was not found. An approximate best match of
System.
Security.Permissions 4.4.0 was resolved.
/app/calculator.csproj: warning NU1701: Package
'Microsoft.AspNet.WebApi.Client 5.2.3' was restored using
'.NETFramework,Version=v4.6.1' instead of the project target
framework '.NETCoreApp,Version=v2.0'. This package may not be
fully compatible with your project.
/app/calculator.csproj: warning NU1701: Package
'Newtonsoft.Json 6.0.1' was restored using '.NETFramework,
Version=v4.6.1' instead of the project target framework
'.NETCoreApp,Version=v2.0'. This package may not be fully
compatible with your project.
/app/calculator.csproj: warning NU1701: Package 'System
4.1.311.2' was restored using '.NETFramework,Version=v4.6.1'
instead of the project target framework '.NETCoreApp,
Version=v2.0'. This package may not be fully compatible with
your project.
Generating MSBuild file /app/obj/calculator.csproj.
nuget.g.props.
Generating MSBuild file /app/obj/calculator.csproj.nuget.g.
targets.
Restore completed in 4.79 sec for /app/calculator.csproj.
```

Una vez creada la imagen, podemos lanzar un contenedor de tipo calculator:

```
docker run -d --name testcalcul -e
URLOptimizerJobs=http://10.0.2.15:5004/api/Jobs/ calculator
```

 La dirección IP a utilizar es la de la máquina que expone el contenedor lanzado inicialmente y basada en la imagen optimizer. A continuación veremos cómo no tener que depender de este valor. Atención, no olvide el símbolo "/" al final de la URL, porque el programa lo necesita para componer las URL unitarias de descripción de los jobs. Se podría hacer evolución automáticamente, pero este no es el caso en el ejemplo proporcionado en la versión 1.0. La utilización de la variable de entorno en la ejecución del contenedor no es realmente útil, porque habíamos fijado un valor por defecto similar en Dockerfile.


Si sigue existiendo el contenedor lanzado anteriormente, la instancia de calculator va a tomar en cuenta los jobs durante el envío de la misma consulta de ejemplo a optimizer y los logs mostrarán esta actividad:

```
29 jobs(s) remaining
28 jobs(s) remaining
27 jobs(s) remaining

[...]

2 job(s) remaining
1 job(s) remaining
No more jobs !
No more jobs !
No more jobs !
No more jobs !
```

Si no se ha sobrepasado el plazo de espera, el servicio `optimizer` devolverá la mejor fecha de reembolso.

 Si el cálculo solicitado implica muchas simulaciones o su ordenador es demasiado lento, no dude en aumentar el valor del plazo de espera en el código fuente `MortgageController.cs`, cambiando el valor de `inicio.AddMinutes(1)` y recompilando la imagen para relanzar el contenedor actualizado.

Una vez que se realiza el cálculo, una petición de `http://localhost:5004/api/Jobs` expondrá la lista de los 29 trabajos con la propiedad `Done`, cambiada a `true`, lo que muestra que todos los jobs se han resuelto.

Siendo rápido (o mejor, aumentando la duración del préstamo para tener más tiempo para realizar la interacción), es posible lanzar dos contenedores en la misma imagen de `calculator`. Entonces comprobaremos, visualizando los logs, que el número de jobs restantes disminuye casi el doble de rápido, lo que prueba que la paralelización se hace correctamente. Volveremos más adelante en detalle sobre esta característica, pero ya podemos señalar que no necesitamos relanzar nada, ni siquiera avisar al servicio `optimizer` que tiene un servicio adicional para resolver los jobs que expone.

## 4. Implementación de enlaces entre contenedores

Durante la ejecución del primer servicio, se dio una observación sobre la complejidad de gestionar sus puertos manualmente y que al final de algunos servicios, era más fácil dejar que Docker

gestionase su plan de asignación. Pero imaginemos ahora que hemos lanzado el primer servicio de la siguiente manera:

```
docker run -d -P optimizer
```

Como se ha mostrado anteriormente, se debería utilizar `docker ps` para encontrar el puerto público expuesto, por ejemplo 32768. Y consecuentemente, pasar la variable de entorno habría sido obligatorio para el segundo servicio:

```
docker run -d --name testcalcul -e  
URLOptimizerJobs=http://10.0.2.15:32768/api/Jobs/ calculator
```

Como este puerto va a cambiar en cada ejecución, va a ser más complicado automatizar todo esto en un script, porque sería necesario que el script busque en los resultados de `docker ps` para recuperar el puerto e insertarlo de manera dinámica en el siguiente comando. Es posible, pero complejo. Además, la máquina host cambiará de IP, lo que hace que la variable de entorno sea doblemente volátil.

Afortunadamente, existe una manera de proceder mucho más sencilla que consiste en utilizar enlaces (*links* en inglés) entre contenedores, funcionalidad ofrecida por Docker. Creando un enlace entre los dos contenedores durante la ejecución del segundo, vamos a dejar que Docker se encargue de la comunicación con el primero. Podemos dar el puerto interno del primer contenedor (y que no se mueve, porque está fijo en su `Dockerfile` a 80) y Docker configurará una configuración de sistema (variables de entorno, introducida en el archivo `hosts`, etc.) que hará que el primer contenedor sea visto por el segundo como una máquina en su red, con el nombre que le hayamos especificado en el enlace.

La línea de comando es la siguiente:

```
docker run -d --name testcalcul -e  
URLOptimizerJobs=http://optim1:80/api/Jobs/ --link  
testoptim:optim1 calculator
```

El argumento `--link` permite especificar que el contenedor llamado `testoptim` (que hemos lanzado previamente), va a ser visible en él durante su ejecución, como una máquina llamada `optim1`. De repente, cambiamos también la variable de entorno para que apunte más a la IP de la máquina host, pero justamente sobre este alias `optim1` (Docker añade automáticamente una entrada en la lista de alias de máquinas). Y, en lugar de tener que




exponer el puerto como puerto público y especificar el valor en esta segunda ejecución, se puede utilizar directamente el puerto interno, lo que es más sencillo (y absolutamente imprescindible en una arquitectura con decenas de contenedores). Además de que el puerto no cambiará más y el nombre del alias se especifica durante la ejecución, la variable de entorno es inútil porque la URL no va a cambiar más.

Desde un punto de vista gráfico, hemos sustituido la ruta compleja A (exposición de un puerto por `-P`, recuperación de este número, asignación de la variable de entorno para apuntar sobre el puerto correcto) por esta, mucho más directa, llamada B (enlace de contenedor a contenedor), en el siguiente diagrama:


 images/esquema24.png

Como el puerto público ya no es útil, es posible lanzar el primer contenedor sin la opción `-P`, porque ya no necesitamos exponer el puerto a las llamadas externas: la información `EXPOSE 80` del `Dockerfile`, es suficiente para Docker para conectar correctamente los contenedores juntos (y hemos visto que para este puerto en particular, no era imprescindible). Por el contrario, esta simplificación solo se podrá realizar al final del viaje, cuando ya no necesitemos probar los contenedores uno por uno.

 Además del alias, Docker crea automáticamente las variables de entorno que retoma la información de los enlaces. Volvemos a dirigir al lector a la documentación detallada de Docker sobre este punto, la gestión de los alias era la mejor práctica para utilizar los enlaces entre contenedores.

Ahora utilizaremos este modo para enlazar los contenedores. La consecuencia de esto es que ya podemos cambiar el valor por defecto de la variable de entorno en el `Dockerfile` del servicio `calculator` (no olvidar reconstruir la imagen después de esta modificación, de lo contrario, lo siguiente no funcionará):

```
FROM aspnetbase:beta4
ADD /src /app
WORKDIR /app
RUN ["dnu", "restore"]
ENV URLOptimizerJobs=http://optimizer:5004/api/Jobs/
CMD ["dnx", ".", "run"]
```

 Eventualmente podremos cambiar el valor por defecto en el código fuente `Program.cs` y hacer que apunte a `localhost`,

porque este programa por supuesto, se puede utilizar fuera de Docker y este enfoque, sería el más lógico. Por el contrario, en la variable de entorno del Dockerfile utilizamos `optimizer`, visto que es el nombre por defecto del contenedor que se lanzará. Por supuesto, el valor `optim1` (o cualquier otro) sigue siendo válido, desde el momento en que es idéntico al utilizado por el alias en la opción `--link`.

Al final, la manera más limpia de lanzar los dos contenedores enlazados, es la siguiente:

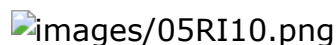
```
docker run -d -P --name testoptim optimizer
docker run -d --name testcalcul --link testoptim:optim1 calculator
```

La opción `-P` solo es útil porque vamos a utilizar Postman para lanzar nuestras pruebas. El comando `docker ps` permite ver sobre qué puerto es necesario lanzar el comando de prueba:

```
jpg@Ubuntu1410:~/mortgage/calculator$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
fe2e40e62d79	calculator:latest	"dnx . run"	6 seconds ago
6d14cbb03709	optimizer:latest	"/bin/sh -c 'sleep 1	30 seconds ago
Up 30 seconds	0.0.0.0:32772->80/tcp	testoptim	

La obtención de este puerto nos permite lanzar nuestra prueba, que esta vez nos devuelve una fecha "real":



La consulta de las trazas nos permite validar que todo se ha desarrollado como esperábamos:

```
jpg@Ubuntu1410:~/mortgage/calculator$ docker logs testoptim
Started
29 jobs created
Index of best repayment month is 210
```


```
jpg@Ubuntu1410:~/mortgage/calculator$ docker logs testcalcul
Mortgage calculation service listening to
http://optim1:5004/api/Jobs/
No more jobs !
29 job(s) remaining
28 job(s) remaining
27 job(s) remaining
[...]
```

```
2 job(s) remaining
1 job(s) remaining
No more jobs !
```

## 5. Detalles del servicio reporting

### a. Funcionamiento

Después de crear dos servicios en .NET, cambiamos de lenguaje: el servicio `reporting` se escribirá en Python y utilizará el framework Flask que permite crear muy fácilmente los API REST, así como la librería `reportlab` para crear un archivo PDF.

 Entre los diferentes frameworks y lenguajes implementados en algunos servicios mostrados en la aplicación de ejemplo, a pesar de que el autor tiene una experiencia más importante en otros lenguajes, Python es el que ha permitido establecer más rápida y sencillamente un API funcional. Es particularmente recomendado para el prototipado rápido.

El código es tan compacto que es posible mostrarlo aquí íntegramente:

```
from flask import Flask
from flask import make_response
from flask import request
from reportlab.pdfgen import canvas

import requests, json
app = Flask(__name__)

@app.route('/pdf', methods=['POST'])
def pdf():

    # Calling mortgage optimizer (in order to decouple the
    # servicios,
    # this would be done in a message queue)
    encabezados = {'Content-type': 'application/json', 'Accept':
    'text/plain' }
    callURL = "http://optimizer"
    callURL += "/api/Mortgage/BestRepaymentDate"
    callURL += "?AmountRepaid=" + request.args['amountRepaid']
    callURL += "&ReplacementFixRate=" +
    request.args['replacementFixRate']
    r = requests.post(callURL, data=request.data,
    headers=encabezados)
    print("Best repayment date is: " + r.content.decode('utf-8'))

    # Generating PDF report
    import io
    output = io.BytesIO()
    p = canvas.Canvas(output)
```

```

    p.drawString(100, 100, 'Best date to reimburse your mortgage is:
' + r.content.decode('utf-8'))
    p.showPage()
    p.save()
    pdf_out = output.getvalue()
    output.close()

    # Sending PDF as a MIME content
    response = make_response(pdf_out)
    response.headers['Content-Disposition'] = "attachment;
filename=report.pdf"
    response.mimetype = 'application/pdf'
    return response

if __name__ == '__main__':
    app.run(host='0.0.0.0')

```

Este archivo, ubicado directamente en la raíz de `reporting`, es auto-suficiente. El único archivo adicional es el llamado `requirements.txt`, y que es una sencilla lista de las dependencias a cargar:

```

swarm@node1 ~/mortgage/reporting $ cat requirements.txt
Flask
reportlab
requests

```

El código fuente mostrado más atrás, expone un API en la ruta llamada `pdf`, que será accesible en modo `POST` y aceptará un contenido JSON que describe el préstamo a tratar, así como los dos argumentos de reembolso ya explicados. El servicio `reporting` enruta esta información al servicio `optimizer`, y recupera el resultado para crear un contenido PDF que enviará a su llamador (type MIME `application/pdf`).

Observe que utilizamos la potencia de los enlaces para relacionar los contenedores, y esta es la razón por la que nos permitimos almacenar una URL en duro en el código (con el nombre de enlace `"optimizer"` en lugar de `"optim1"`, como antes): esta se relaciona con el contenedor correcto durante la ejecución. Si se supone que este código Python tenía que funcionar fuera de un contenedor como el servicio de cálculo, sería necesario hacer parametrizable este valor.

La última parte del código fuente abre el API en el host actual.

## b. Dockerización

El archivo `Dockerfile` es un poco menos compacto que los anteriores, pero es muy sencillo:

```
FROM python:3.6
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
COPY webtest.py ./
CMD ["python", "-u", "./webtest.py"]
```

A título puramente informativo, a continuación se muestra el contenido del Dockerfile de la primera edición de este libro:

```
FROM ubuntu:trusty
MAINTAINER jp.gouigoux@free.es

RUN apt-get update \
    && apt-get install -y \
        python \
        python-pip \
        wget \
    && pip install Flask

RUN apt-get install -y python-reportlab

COPY src/ .

EXPOSE 5000
ENTRYPOINT ["python", "-u"]
CMD ["webtest.py"]
```

Los cambios están principalmente en la imagen básica, antes una Ubuntu sobre la que hemos instalado Python y las dependencias necesarias. Desde ahora, se trata de una imagen “oficial” de Python. Se utiliza la herramienta `pip` para instalar las dependencias, pero estas ya no están en duro en el archivo Dockerfile. Para terminar, `ENTRYPOINT` y `CMD` ya no se separan ya que no proponemos archivo de script alternativo; no es útil apuntar a otro archivo durante la ejecución.

Al contrario que el resto de imágenes, donde todo el código fuente se almacena en un directorio `src` que copiamos en la imagen, ubicamos el único archivo de código fuente en la raíz de esta imagen, vista su compacidad. Se expone en el puerto 5000. Observe que el código fuente de Python no hace referencia a un puerto particular: el programa generado de esta manera no se restringirá a un único puerto y escuchará todo el flujo entrante.

Para la ejecución de Python propiamente dicha durante la puesta en marcha de un contenedor, `ENTRYPOINT` tiene el comando de ejecución del proceso `python` con la opción `-u`, que es imprescindible en modo depuración, porque impide que los logs se almacenen en un búfer. Sin esta opción, la visualización de los logs puede reservar sorpresas, con entradas que permanecen en

los búfers para mejorar el rendimiento, cuando nos gustaría verlas en los logs. El vínculo <http://stackoverflow.com/questions/29663459/python-app-does-not-print-anything-when-running-detached-in-docker> explica en detalle este modo de funcionamiento.

El segundo argumento del comando `python` es el nombre del archivo a lanzar. En efecto, Python es un lenguaje en modo interpretado y no compilado. La aplicación a llamar es el proceso `python`. Además, el script `webtest.py` que hemos creado se le debe pasar como argumento, para que se carguen y se interpreten sus comandos.

### c. Pruebas


En este estado de la implementación de la aplicación ejemplo, empezamos a tener un conocimiento importante. Vamos a dar solo los comandos para, una vez compilada la imagen, lanzar un nuevo contenedor con el nombre que hemos elegido para las captura de pantallas y recuperar el puerto para las pruebas:

```
jpg@Ubuntu1410:~/mortgage/reporting$ docker run -d --name testreport
-P --link testoptim:optimizer reporting
c33607d2e0064687b6e6cd1cca561eff62287b3ef017ac4542058f6b2b209c52
jpg@Ubuntu1410:~/mortgage/reporting$ docker ps
```

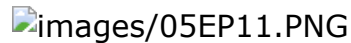
CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED
c33607d2e006	reporting:latest	"python -u webtest.p	testreport	About a minute ago
Up About a minute	0.0.0.0:32773->5000/tcp			
fe2e40e62d79	calculator:latest	"dnx . run"	testcalcul	27 minutes ago
Up 27 minutes				
6d14cbb03709	optimizer:latest	"/bin/sh -c 'sleep 1	testoptim	28 minutes ago
Up 28 minutes	0.0.0.0:32772->5004/tcp			

Docker ha asignado el puerto 32773, y nuestra prueba consiste en llamar en Postman a `http://localhost:32773/pdf?amountRepaid= 20000& replacementFixRate=0.025` en modo POST, con el mismo JSON en el cuerpo de la petición anterior.

Después de algunos segundos de cálculo, la herramienta propone hacer una copia de seguridad de lo que se ha devuelto en un archivo PDF, y muestra el resto de la respuesta:

images/05RI11.png

Un vistazo al archivo PDF muestra un informe extremadamente en bruto, pero con el contenido esperado:



Por lo tanto, estamos preparados para pasar a la siguiente etapa de nuestra pila micro-servicios, a saber, la creación de `notifier` que va a utilizar el informe generado para enviarlo en un mail al destinatario designado.

## 6. Detalles del servicio notifier

### a. Funcionamiento

No podemos recorrer las principales plataformas de desarrollo de servicios sin pasar por Java EE. Lo haremos con el servicio `notifier`, cuya responsabilidad es informar a la persona de los resultados del análisis.

Los defectos de Java están en sus cualidades, a saber, la ampliación de los métodos para realizar un API REST, algunas veces hace difícil la elección. El presente ejemplo muestra el uso del framework Spark, mantenida por su gran simplicidad de implantación.

Mostraremos de nuevo la arborescencia destino, porque al contrario que los servicios anteriores, no es sencillo:

```
notifier/  
├── Dockerfile  
├── pom.xml  
└── src/  
    ├── main/  
    │   └── java/  
    │       └── sparkexample/  
    │           ├── Hello.java  
    │           └── MailSender.java
```

El código fuente se basa en el ejemplo proporcionado con Spark, de ahí que finalmente el nombre encaje muy bien con esta pequeña aplicación ejemplo.

Después de las importaciones, el archivo `Hello.java`, que tiene el código siguiente, arranca la aplicación poniéndola a la escucha en el puerto 5000, y asociando la llamada en modo `POST` en la ruta `/repayment` a una operación contenida en el código fuente siguiente, en la última llave abierta, que no detallaremos:

```
public class Hello {  
  
    public static void main(String[] args) {  
        setPort(5000);  
  
        post("/repayment", (request, response) -> {
```

A groso modo, el código fuente devuelve un error si no encuentra las variables de entorno SMTP\_AUTH\_LOGIN y SMTP\_AUTH\_PASSWORD (necesarias para enviar un mail), y después llama al servicio reporting con el contenido que se le ha pasado a sí mismo (parecido a reporting que solo pasa el contenido obtenido a optimizer). Para terminar, compone un mail con el archivo PDF como archivo adjunto y lo envía. Esta parte del código se delega a la clase MailSender, contenida en el segundo archivo de código fuente del mismo nombre, que usa el sufijo .java.

Lo fundamental del código fuente de MailSender.java, se ha copiado del artículo en <http://www.mkkyong.com/java/javamail-api-sending-email-via-gmail-smtp-example/>. En esta clase es donde debe intervenir si no dispone de una cuenta Gmail o desea utilizar otro proveedor para enviar mensajes electrónicos. En el caso contrario, debe configurar su identificador y su contraseña en las variables de entorno citadas anteriormente, para que la prueba funcione. Se ha hecho un esfuerzo para poner en la parte superior del archivo todas las propiedades sobre las que el lector puede intervenir:

```
Properties props = new Properties();  
props.put("mail.smtp.auth", "true");  
props.put("mail.smtp.starttls.enable", "true");  
props.put("mail.smtp.host", "smtp.gmail.com");  
props.put("mail.smtp.port", "587");
```

Para terminar, como vamos a utilizar Maven para compilar el código fuente Java, es necesario crear un archivo pom.xml en el que declaremos las dependencias. El ejemplo que viene en la documentación de Spark, se ha ampliado para añadir las dependencias correspondientes a nuestros componentes adicionales, para llamar a un servicio, manipular los archivos y enviar un mail. Al final, el inicio del archivo pom.xml es como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```



```
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>hellodocker</groupId>
  <artifactId>hellodocker</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>com.sparkjava</groupId>
      <artifactId>spark-core</artifactId>
      <version>2.0.0</version>
    </dependency>
    <dependency>
      <groupId>javax.mail</groupId>
      <artifactId>mail</artifactId>
      <version>1.4.7</version>
    </dependency>
  </dependencies>
```

Una vez que se ha creada la arborescencia con los archivos correctos, podemos pasar a la construcción de la imagen.

## b. Dockerización

La escritura del archivo `Dockerfile`, será más relajante que la del código Java y `pom.xml`, más concretamente. Existe una imagen básica oficial para los desarrollos Java, lo que nos va a hacer ahorrar mucho tiempo, como con la imagen ASP.NET ofrecida por Microsoft.

El `Dockerfile` (también tomado del ejemplo introductorio de Spark), se presenta a continuación:

```
FROM java:8

RUN apt-get update && apt-get install -y maven

WORKDIR /code


ADD pom.xml /code/pom.xml
RUN ["mvn", "dependency:resolve"]
RUN ["mvn", "verify"]

ADD src /code/src
RUN ["mvn", "package"]

EXPOSE 5000
CMD ["/usr/lib/jvm/java-8-openjdk-amd64/bin/java","-jar",
"target/sparkexample-jar-with-dependencies.jar"]
```

La directiva `FROM` muestra claramente que utilizamos una imagen preparada con el entorno de compilación y de ejecución Java, en versión 8.

El primer comando `RUN` instala Maven. A continuación, se llama a este último para resolver las dependencias. Para terminar, después de añadir el código fuente en la imagen, el comando `mvn package` provoca la compilación del código fuente Java. El puerto expuesto será el 4567 y el comando de ejecución será `java -jar [el paquete JAR generado justo antes]`.


 Durante la compilación de la imagen, ármese de paciencia: Maven va a descargar varias decenas de MB de dependencias en cascada y, en función de su ancho de banda, esto pueda llevar cierto tiempo. La ligereza es el punto fuerte de Python, pero no el de Java.

Una vez compilada la imagen, vamos a poder probar la llamada del servicio expuesto por Java.

### c. Pruebas

A este efecto, lanzamos el contenedor con un enlace sobre `reporting`, por lo que necesita `notifier`, así como los valores de los argumentos necesarios para el envío de mail:

```
docker run -d --name testnotif -P -e
SMTP_AUTH_LOGIN=su.identificador@gmail.com -e
SMTP_AUTH_PASSWORD=su.contraseña@gmail --link
testreport:reporting notifier
```

 La definición de las contraseñas en las variables de entorno en memoria, por lo tanto sin soporte de un archivo, es uno de los medios más seguros de pasar información confidencial.

Por defecto, Gmail no permite a todo el mundo utilizar el servidor de envío de mails, principalmente para limitar los riesgos de que las cuentas personales, fácilmente pirateables, se conviertan en vectores spam. Por lo tanto, para que nuestro ejemplo funcione, es necesario dar autorización a los clientes diferentes de Gmail para enviar mails en nombre de su cuenta.

Para esto, siga las instrucciones que se especifican en la dirección <https://www-google-com.sabidi.urv.cat/settings/security/lesssecureapps> para autorizar las aplicaciones menos seguras.


Si esta opción no está activada, el mensaje de error tiene la ventaja de ser muy útil, haciendo referencia a una página de

soporte explícita:

```
javax.mail.AuthenticationFailedException: 534-5.7.14
[...]
534-5.7.14 X_LRE0ywxKApI3MV7LXw9y--bL5g> Please log in via your
534-5.7.14 web browser and then try again.
534-5.7.14 Learn more at
534-5.7.14 https://support.google.com/mail/bin/answer.py?
answer=78554
gl4sm23439276wjs.47 - gsmtip
```


Una vez que se lanza el contenedor (y se localiza su puerto público utilizando el comando `docker ps`), y con la idea de mostrar un método alternativo a Postman, esta vez vamos a lanzar nuestra prueba directamente en la línea de comandos, utilizando las opciones de `curl`:

```
curl
-d '{"startMonth": 13,"repaymentLengthInMonths": 200,
"amountBorrowed": 20000,"isRateFixed": true,"fixRate": 0.035}'
-X POST
-H "Content-Type: application/json"
http://localhost:32782/repayment?email=jp.gouigoux@free.es&repaid
=20000&newRate=0.025
```

 Atención, los nombres de los argumentos a utilizar en la URL no son los mismos que antes. Esto se ha hecho de manera expresa para mostrar los casos habituales de error.

Cuanto más complejo sea el código de este servicio, más sencilla será la validación de la prueba: es suficiente con comprobar que el destinatario, cuya dirección se ha indicado en el argumento de la petición que se pasa a `curl`, ha recibido correctamente el mensaje. Observe que se trata de la dirección que se pasa en el argumento de la URL llamado `email`, y no la dirección de correo correspondiente al emisor.

Esto será lo que aparezca como emisor:

 images/05RI12.png

Recientemente, Google ha reforzado todavía más la seguridad en el envío de mails por un cliente diferente de Gmail, incluso si está autorizado para el uso de clientes menos seguros, es perfectamente posible que reciba este tipo de mensajes durante la primera prueba:



Pulsando en el enlace propuesto, puede encontrar la actividad "sospechosa" y reconocerla como autorizada en el sitio web asociado:



Evidentemente, se trata de comprobar que esta actividad es la suya. La fecha y hora le deberían dar una buena indicación, pero si sospecha aunque sea ligeramente (como cualquier buen desarrollador preocupado por la seguridad), lo mejor es validar también que la IP indicada es la de la máquina que usted utiliza.

En el ejemplo mostrado anteriormente, se trataba de una máquina en el cloud West Europe de Azure cuya dirección se corresponde con la indicada en el portal Azure:



Toda la parte "back office" (todas las operaciones de negocio realizadas por administradores, como el cálculo de préstamos y la creación de un documento binario) de nuestra aplicación era funcional. Lo que continúa, afecta al "front office" (los servicios expuestos a los usuarios finales, normalmente la interfaz web y la gestión del almacén de una petición de informe). Sin embargo, antes de mostrar la construcción del portal web, vamos a detenernos en una imagen secundaria (o quizás "herramienta"), que utilizará el portal, es decir, el contenedor, para la base de datos.

## 7. Detalles del servicio de persistencia

Con el objetivo de conservar las definiciones de préstamos, de manera que se pueda lanzar un informe sobre la mejor fecha de reembolso mucho tiempo después de haber creado el directorio de préstamo, es importante disponer de un servicio de persistencia. La base de datos NoSQL MongoDB se ha elegido por su simplicidad extrema de implantación y utilización en el marco de las tecnologías web.

Esta vez, ni imagen básica ni `Dockerfile`: vamos a utilizar directamente la imagen oficial recuperada de Docker Hub. Sobre todo es una buena práctica no volver a crear las imágenes ya existentes. Cuando existe una imagen oficial, tenemos la


seguridad de que su contenido está perfectamente controlado, bien probado, con todas las configuraciones y garantías de funcionamiento necesarias.

Para este contenedor, tenemos que instanciar la imagen existente:

```
docker run -d -p 27017:27017 --name mongo mongo:3.0.3
```

## 8. Implementación eventual de una imagen básica

Otra imagen "de utilidad" que podríamos necesitar (en caso del portal y en otras partes), es una imagen básica que no se instanciaría como un contenedor, sino únicamente heredada por la imagen (en nuestro ejemplo, portal) que vamos a construir más adelante. El interés de este tipo de imagen es que evita pasar por una descarga de todas las dependencias en cada compilación, la imagen básica sirve de caché para las librerías más estándares. Es cierto que el hecho de separar la carga de las dependencias y el soporte del código fuente, permite beneficiarse de un nivel de caché por el mecanismo de generación de las imágenes por capas, incluido en Docker. Pero algunas veces, varias imágenes utilizan dependencias comunes. En este caso, puede ser interesante crear una imagen básica en la que se carguen las dependencias comunes y después, las imágenes se basen en esta y lleguen a especializarse no solo por el código fuente, sino por algunas dependencias adicionales cuando sea necesario. De esta manera, incluso en el caso de añadir dependencias, solo será necesaria una parte de las descargas.

 Otra optimización, en el caso de lenguajes interpretados y no compilados, es utilizar un volumen para que la imagen apunte al código fuente externo. De esta manera, la imagen cambia de comportamiento en función de la evolución del código fuente y esto sin la menor operación de compilación. Esta ventaja se contrarresta por el hecho de que la imagen ya no es autónoma. Al final, este modo de funcionamiento es interesante en desarrollo, pero una vez en producción, es necesario utilizar imágenes autosuficientes, completas e inmutables.

Si implementamos este tipo de imagen bajo el nombre de nodebase, solo necesitaríamos un único archivo que defina el proyecto, así como el `Dockerfile`. En la tecnología Node.js que vamos a utilizar, este archivo de proyecto se llama

package.json, y para nuestra imagen básica, solo contendría las pocas líneas siguientes:

```
{
  "name": "nodebase",
  "private": true,
  "version": "0.0.1",
  "description": "Empty application, used to create a Docker base
image for Node",
  "author": "JP Gouigoux <jp.gouigoux@free.es>",
  "dependencies": {
    "express": "~4.16.2",
    "mongoose": "~4.12.2"
  }
}
```

El archivo Dockerfile se presenta como sigue:

```
FROM node:8.7-alpine
ADD package.json
RUN npm install
```

La llamada de `npm install` va a lanzar todas las dependencias desde Internet, basándose en el archivo `package.json` copiado en la raíz de la imagen. Esto utilizará las librerías `express` (gestión de las rutas, principalmente, en nuestro caso) y `mongoose` (acceso simplificado a MongoDB).

La compilación de esta imagen llevará algo de tiempo, pero al menos ya no tendremos que esperar la descarga de todas estas librerías, cada vez que volvamos a compilar la imagen `portal`. Para dar una idea del tiempo y el ancho de banda ganados, a continuación se muestra la salida de la etapa de compilación:

```
jpg@Ubuntu1410:~/mortgage/nodebase$ docker build -t notifier .
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0: FROM node:0.12.4
---> 3575f1347ce7
Step 1: ADD package.json /tmp/package.json
---> ee28e63d07eb
Removing intermediate container b74053acda28
Step 2: RUN cd /tmp && npm install
---> Running in 3dd11d4421ef

> kerberos@0.0.3 install
/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/kerberos
> (node-gyp rebuild 2> builderror.log) || (exit 0)

make: Entering directory
'/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/kerberos
/build'
SOLINK_MODULE(target) Release/obj.target/kerberos.node
SOLINK_MODULE(target) Release/obj.target/kerberos.node: Finished
```

```

COPY Release/kerberos.node
make: Leaving directory
'/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/kerberos
/build'

> bson@0.2.2 install
/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/bson
> (node-gyp rebuild 2> builderror.log) || (exit 0)

make: Entering directory
'/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/bson/bui
ld'
CXX(target) Release/obj.target/bson/ext/bson.o
bson.target.mk:82: recipe for target
'Release/obj.target/bson/ext/bson.o' failed
make: Leaving directory
'/tmp/node_modules/mongoose/node_modules/mongodb/node_modules/bson/bui
ld'
express@4.7.4 node_modules/express
+-- merge-descriptors@0.0.2
+-- utils-merge@1.0.0
+-- cookie@0.1.2
+-- escape-html@1.0.1
+-- cookie-signature@1.0.4
+-- range-parser@1.0.0
+-- fresh@0.2.2
+-- vary@0.1.0
+-- finalhandler@0.1.0
+-- qs@0.6.6
+-- media-typer@0.2.0
+-- parseurl@1.2.0
+-- methods@1.1.0
+-- serve-static@1.4.4
+-- buffer-crc32@0.2.3
+-- path-to-regexp@0.1.3
+-- depd@0.4.4
+-- debug@1.0.4 (ms@0.6.2)
+-- accepts@1.0.7 (negotiator@0.4.7, mime-types@1.0.2)
+-- type-is@1.3.2 (mime-types@1.0.2)
+-- proxy-addr@1.0.1 (ipaddr.js@0.1.2)
+-- send@0.7.4 (ms@0.6.2, mime@1.2.11, finished@1.2.2)

mongoose@3.6.20 node_modules/mongoose
+-- regexp-clone@0.0.1
+-- sliced@0.0.5
+-- muri@0.3.1
+-- hooks@0.2.1
+-- mpath@0.1.1
+-- ms@0.1.0
+-- mpromise@0.2.1 (sliced@0.0.4)
+-- mongodb@1.3.19 (kerberos@0.0.3, bson@0.2.2)
---> 58ea84bfae7f
Removing intermediate container 3dd11d4421ef
Successfully built 58ea84bfae7f

```

Las librerías Node.js no son voluminosas, pero como tiran de las dependencias en cascada, de repente nos encontramos rápidamente con numerosos archivos recuperados, como se puede ver anteriormente.

Esta sección solo era a título informativo, porque en un proyecto tan sencillo como el utilizado para nuestra aplicación de ejemplo, es claramente inútil establecer una imagen básica. El nivel de caché operada por la generación de imágenes de Docker, ya es ampliamente suficiente. La imagen `portal` que se describe a continuación, se basará directamente en la imagen oficial `node`.

## 9. Detalles del servicio portal

### a. Funcionamiento

El servicio `portal` utiliza principalmente la tecnología Node.js, que gestiona los servicios web. Node.js es - para hacerlo muy corto - un medio de realizar un servidor utilizando JavaScript, lenguaje utilizado tradicionalmente en el lado cliente, pero que es conveniente finalmente para la gestión de servidores asíncronos y con alto rendimiento. Esta primera funcionalidad se corresponde con la base de la arborescencia del proyecto, que se muestra a continuación:

```
portal/
├── Dockerfile
├── index.js
├── package.json
├── public/
│   ├── app.js
│   └── index.html
└── README.md
```

También utilizaremos este servicio para servir archivos estáticos que se leerán e interpretarán en el lado cliente. Para marcar correctamente la separación, estos archivos se ubicarán en el directorio público. En el código Node.js, que se encuentra en `index.js`, un comando permitirá servir estos archivos de manera estática:

```
app.use(express.static(__dirname + '/public'));
```

Además, cuando se llame al servidor con cualquier URL, se redirigirá a la página `index.html`:

```
app.get('*', function(req, res) {
  res.sendFile('./public/index.html ');
});
```

Se observa que los archivos estáticos también contienen una parte de código, escrita en JavaScript, pero que se ejecutará en



el lado cliente. Esta porción de código utiliza las librerías Angular.js (para la organización MVC de las páginas) y Bootstrap (para el estilo y el layout). El código HTML es muy explícito y el archivo `app.js` que contiene el JavaScript del lado cliente, ya no plantea dificultades en particular. Simplemente mostraremos el modo Angular.js que permite conectar la página a las llamadas AJAX de los API expuestos por Node.js:

```
angular.module('mortgageSampleApp')
.factory('Mortgages', ['$http', function($http) {
  return {
    get: function() {
      return $http.get('/api/mortgages');
    },
    report: function(reportData) {
      return $http.post('/api/report', reportData);
    },
    create: function(mortgageData) {
      return $http.post('/api/mortgages', mortgageData);
    }
  };
}]);
```

Para volver al servidor Node.js, y en particular a los API sobre los que acabamos de hablar, este expone principalmente un API que permite recuperar la lista de préstamos y crea ahí, enlazado por Mongoose con la base de datos MongoDB:

```
app.get('/api/mortgages', function(req, res) {
  getMortgages(res);
});
app.post('/api/mortgages', function(req, res) {
  MortgageModel.create({
    text: req.body.text,
    startMonth: req.body.start,
    repaymentLengthInMonths: req.body.duration,
    amountBorrowed: req.body.amount,
    isRateFixed: req.body.isRateFixed,
    fixRate: req.body.rate
  }, function(err, mortgage) {
    if (err)
      res.send(err);
    getMortgages(res);
  });
});
```

Pero el servidor también expone un API para llamar a la creación del informe sobre el mejor momento para reembolsar el préstamo, designado por el argumento `Reference` que se le pasa:


```
app.post('/api/report', function(req, res) {
  MortgageModel.findOne({ 'text': req.body.reference },
function(err, mortgage) {
```

```

    var contentString = JSON.stringify(mortgage);
    var headers = {
        'Content-Type': 'application/json',
        'Content-Length': contentString.length
    };
    var options = {
        host: 'notifier',
        port: 5000,
        path: '/repayment?email=' + req.body.email +
        '&repaid=' + req.body.repaid + '&newRate=' + req.body.newRate,
        method: 'POST',
        headers: headers
    };
    var repaymentRequest = http.request(options);
    repaymentRequest.write(contentString);
    repaymentRequest.end();
  });
});

```

La implementación de este API, como se puede comprobar, consiste principalmente en recuperar la definición completa del préstamo desde el modelo de datos y después, llamar al servicio `notifier` al que se le ha delegado el envío del mail con el informe.

 Con el objetivo de mejorar la separación de las responsabilidades, se podría argumentar que sería más elegante llamar a un servicio de generación del informe, y después pasarlo al servicio `notifier`, en lugar de cargar explícitamente la generación del informe, lo que crea un acoplamiento en la conducta. La estructura perfecta de una arquitectura basada en micro-servicios es un tema de punto de vista y varía en función de los casos principales de uso.

## b. Dockerización

El contenido del archivo `Dockerfile`, es el siguiente:

```

FROM node:8.7-alpine
ADD package.json .
RUN npm install
RUN mkdir -p /opt/app && cp -a /node_modules /opt/app/

ADD index.js .
ADD public/ public/
EXPOSE 8080
CMD ["node", "index.js"]

```

No hay mucho que explicar sobre este archivo, excepto que la aplicación se sitúa en el directorio `/opt/app` y que la manera estándar de arrancar un servidor Node.js es lanzar el comando

node, utilizando como argumento el archivo principal, a saber `index.js`. Se ha elegido el puerto 8080 como puerto expuesto, lo que se manifiesta en el código de `index.js`, en las líneas siguientes:

```
var port = process.env.PORT || 8080;
app.listen(port);
console.log("App listening on port " + port);
```

La variable de entorno `PORT` se podrá utilizar si es necesario, para cambiar el puerto pero, como se ha mostrado más atrás, lo sencillo es utilizar la redirección del puerto para esto. La primera línea tomará 8080 como valor por defecto, que es nuestro caso.

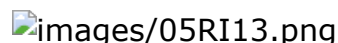
Seguidamente, la imagen se puede construir con ayuda del comando habitual. Acabamos de terminar la composición de los diferentes servicios de nuestra arquitectura y vamos a poder pasar la prueba final de integración completa de la solución.

### c. Pruebas

Para probar esta vez, no hay necesidad de recurrir a herramientas como Postman: vamos simplemente a llamar a `http://localhost:8080`, una vez se lance el contenedor en esta imagen, con la dirección de puerto siguiente:

```
docker run -d --name testportal --link mongo:mongo --link
testnotif:notifier -p 8080:8080 portal
```

El navegador mostrará la siguiente página:



A continuación podremos probar que los datos se almacenan correctamente y que la llamada de la generación del informe, indicando el identificador del directorio de préstamo creado previamente, provoca la recepción de un mail. No volvemos a explicar la manera de hacer esto, ya que se ha descrito al inicio del capítulo.

## 10. Estado alcanzado

Después de la sección para presentar la aplicación, la presente sección de este capítulo ha mostrado cómo crear las imágenes, lanzar los seis contenedores que forman nuestra aplicación de

micro-servicios y relacionarlas de manera que las pruebas muestren un funcionamiento correcto del conjunto.

Al final, un comando `docker ps` permite ver lo que se está ejecutando:

CONTAINER ID	IMAGE	COMMAND	
CREATED			
STATUS	PORTS	NAMES	
fdc832c07e19	portal:latest	"node index.js"	7
minutes ago			
Up 7 minutes	0.0.0.0:8080->8080/tcp	testportal	
89e947d7a674	mongo:3.0.3	"/entrypoint.sh mong	About
an hour ago			
Up About an hour	0.0.0.0:27017->27017/tcp	mongo	
1720d37643ab	notifier:latest	"/usr/lib/jvm/java-8	
22 hours ago			
Up 22 hours	0.0.0.0:32782->5000/tcp	testnotif	
c33607d2e006	reporting:latest	"python -u webtest.p	
24 hours ago			
Up 24 hours	0.0.0.0:32773->5000/tcp	testreport	
fe2e40e62d79	calculator:latest	"dnx . run"	
24 hours ago			
Up 24 hours		testcalcul	
6d14cbb03709	optimizer:latest	"/bin/sh -c 'sleep 1	
24 hours ago			
Up 24 hours	0.0.0.0:32772->80/tcp	testoptim	

Cada servicio con un punto de entrada se expone en un puerto diferente, lo que permite entrar en la aplicación por cualquier punto. En una instalación más estándar, solo se expondría el puerto 8080 porque la interfaz ofrecida por el servicio portal se supone que es el único medio para acceder a la aplicación, o en todo caso, el medio más común y lógico para el usuario.

# Desarrollar automáticamente con Docker Compose

## 1. Principio de Docker Compose

Haciendo abstracción de la construcción de las imágenes y concentrándose en su instanciación en forma de contenedores, el siguiente script retoma el conjunto de comandos que se deben ejecutar para relanzar la aplicación, si el conjunto se acaba de perder (parada de la máquina host, eliminación accidental de los contenedores, etc.):

```
docker run -d -P --name testoptim optimizer
docker run -d --name testcalcul --link testoptim:optim1 calculator
docker run -d --name testreport -P --link testoptim:optimizer
reporting
docker run -d --name testnotif -P -e
SMTP_AUTH_LOGIN=su.identificador@gmail.com -e
SMTP_AUTH_PASSWORD=su.contraseña.gmail --link testreport:reporting
notifier
docker run -d -p 27017:27017 --name mongo mongo:3.0.3
docker run -d --name testportal --link mongo:mongo --link
testnotif:notifier -p
8080:8080 portal
```

Crear un script para realizar todas estas operaciones, no es complicado y permite ganar mucho tiempo durante la siguiente ejecución. Pero también sería necesario crear un script para detener todos los contenedores, otro para eliminarlos, etc. Y durante la próxima implantación de una arquitectura basada en contenedores, sería necesario crear otros scripts diferentes desde el punto de vista de los nombres y las funcionalidades, pero con una estructura finalmente muy próxima.

La tecnología Docker Compose nos va a permitir dedicarnos únicamente a la definición de la arquitectura y dejar la gestión de los scripts a la herramienta. Un archivo `docker-compose.yml` va a contener la lista de los contenedores con sus enlaces, su imagen, la definición de los puertos, etc. y el comando `docker-compose` nos permitirá controlar en una única línea de comandos la ejecución de todos los contenedores descritos en este archivo, su parada, etc.

Para instalar Docker Compose, utilizaremos el método ofrecido por <https://docs.docker.com/compose/install/>, a saber:

```
curl -L
https://github.com/docker/compose/releases/download/1.16.1/docker-
compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose
```

El primer comando recupera el ejecutable correcto y lo deposita en el lugar previsto en la arborescencia Linux. El segundo hace que el archivo sea ejecutable.

## 2. Escritura del archivo docker-compose.yml

El archivo `docker-compose.yml` está disponible directamente en el directorio `mortage`, y su contenido es el siguiente:

```
version: '2'
services:
  mongo:
    image: "mongo:3.2"
  optimizer:
    build: ./optimizer
    image: jpgougoux/optimizer
  calc:
    build: ./calculator
    image: jpgougoux/calculator
    links:
      - optimizer
    environment:
      - "URLOptimizerJobs=http://optimizer/api/Jobs/"
  reporting:
    build: ./reporting
    image: jpgougoux/reporting
    links:
      - mongo
      - optimizer
  notifier:
    build: ./notifier
    image: jpgougoux/notifier
    links:
      - reporting
    environment:
      - SMTP_AUTH_LOGIN=jp.gougoux@gmail.com
      - SMTP_AUTH_PASSWORD=${GMAIL_PWD}
  app:
    build: ./portal
    image: jpgougoux/portal
    links:
      - mongo
    ports:
      - "8080:8080"
```

Se utiliza el formato YAML para describir una lista de contenedores, con un identificador que está escrito al inicio de la línea y las propiedades un poco más abajo. La primera propiedad puede ser imagen o build, dependiendo de si la imagen se

recupera sobre un registro o proviene del directorio citado (de ahí el interés de poner nuestro archivo `docker-compose.yml` en la base del directorio `mortgage`, que contiene el resto de subdirectorios con los archivos `Dockerfile` correspondientes). Como se explica en un capítulo anterior, es posible utilizar a la vez las palabras clave `imagen` y `build`, lo que permite a Docker Compose utilizar directamente la imagen compilada o bien, conocer el directorio a utilizar para compilarla, solicitándolo con el comando `docker-compose build`.

Las propiedades `links` y `environment` no necesitan realmente explicaciones, en el sentido en que ellas solo retoman los valores que habíamos implantado anteriormente en la versión "manual" de la ejecución de nuestra aplicación.

Observe que estamos en la versión 2.0 de la gramática Docker Compose. Al contrario de lo que sucedía con la primera versión que solo lista los contenedores, esta versión permite la definición de servicios que representa un nivel adicional de indirección respecto a los contenedores. Un servicio pueda provocar la implantación de contenedores en localizaciones diferentes de la red creada por Docker para conectarlas. En la versión 1.0 de la gramática Docker Compose, no se soportaba ninguna red, y todos los contenedores se conectaban simplemente en la misma red `bridge`.

De hecho, esta funcionalidad es tan potente que nos permite eliminar completamente la noción de enlace entre los contenedores. En efecto, en la red montada automáticamente por los contenedores de Docker Compose, se van a crear alias para que los contenedores aparezcan bajo su nombre de servicio. También se implementa un sistema de reparto de carga para que, si el servicio se instancia en forma de varios contenedores (comando `docker-compose scale`), las diferentes instancias se llamen por el mismo alias.


Al final, el archivo `docker-compose.yml` es el siguiente:

```
version: '2'
services:
  mongo:
    image: "mongo:3.2"
    optimizer:
      build: ./optimizer
      image: jpgouigoux/optimizer
  calc:
    build: ./calculator
    image: jpgouigoux/calculator
    environment:
```

```
- "URLOptimizerJobs=http://optimizer/api/Jobs/"
reporting:
  build: ./reporting
  image: jpgougoux/reporting
notifier:
  build: ./notifier
  image: jpgougoux/notifier
environment:
  - SMTP_AUTH_LOGIN=jp.gougoux@gmail.com
  - SMTP_AUTH_PASSWORD=${GMAIL_PWD}
app:
  build: ./portal
  image: jpgougoux/portal
ports:
  - "8080:8080"
```

Observe que puede sustituir la variable para la contraseña de mensajería, por la vuestra si está seguro de que el archivo `docker-compose.yml` no terminará nunca en manos de personas con malas intenciones, pero es más seguro proceder como se ha explicado más atrás, y ofrecer una variable de entorno en la shell utilizada durante la ejecución de Docker Compose, para transmitir esta contraseña.

Otra observación es que esta vez, el único puerto que hemos abierto sobre la máquina host es el puerto 8080 del servicio portal. Como habíamos explicado, la utilidad de exponer los puertos del resto de servicios es limitada, ya que solo son secundarios y en ningún caso un punto de entrada para el usuario. Para reducir la superficie de ataque, es prudente no exponerlos.

 Aunque sea posible introducir números de puertos sin comillas, se recomienda utilizarlas porque el formato YAML puede, en algunos casos, interpretar los valores como un número en base sexagesimal. Ver <http://www.fig.sh/yml.html>, para más detalles.

## 3. Implementación

### a. Preparación

Con el objetivo de no tener conflictos (en particular sobre el puerto 8080), vamos a comenzar eliminando todos los contenedores que habíamos arrancado para nuestra aplicación ejemplo. Nuestro objetivo es volver a montar esta arquitectura completa utilizando Docker Compose.

El comando que se debe utilizar (con precaución, porque elimina todos los contenedores activos), es el siguiente:



```
docker rm -fv `docker ps -aq`
```

Con el objetivo de preparar la ejecución de la aplicación, vamos a fijar el valor de las variables de entorno necesarias:

```
export SMTP_AUTH_LOGIN=su.identificador@gmail.com
export SMTP_AUTH_PASSWORD=su.contraseña
```

Como hemos utilizado una variable de entorno `GMAIL_PWD` para el valor por defecto de `SMTP_AUTH_PASSWORD`, también sería posible modificarla y cambiar el identificador directamente en el archivo `docker-compose.yml`.

Es útil lanzar (desde el directorio `mortgage`, de manera que la herramienta encuentre automáticamente el archivo YAML de configuración anteriormente descrita), el siguiente comando, que va a compilar todas las imágenes necesarias:

```
docker-compose build
```

## b. Ejecución de los contenedores

Siempre posicionado en el directorio `mortgage`, la ejecución de la aplicación se hace ahora con el único comando siguiente:

```
docker-compose up
```

Por defecto, el comando muestra las trazas, que Docker Compose agrupa, pero utilizando los identificadores (así como un código de colores en los terminales que lo soportan), que permite reconocer rápidamente de dónde provienen las líneas de información:

```
Creating mortgage_optim_1...
Creating mortgage_calc_1...
Creating mortgage_db_1...
Creating mortgage_report_1...
Creating mortgage_notif_1...
Creating mortgage_app_1...
Attaching to mortgage_optim_1, mortgage_calc_1, mortgage_db_1,
mortgage_report_1, mortgage_notif_1, mortgage_app_1
report_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
db_1      | 2015-06-21T22:18:22.978+0000 I JOURNAL
[initandlisten] journal dir=/data/db/journal

[...]

db_1      | 2015-06-21T22:18:23.873+0000 I NETWORK
[initandlisten] waiting for connections on port 27017
notif_1   | SLF4J: Failed to load
class "org.slf4j.impl.StaticLoggerBinder".
notif_1   | SLF4J: Defaulting to no-operation (NOP)
```

```

logger implementation
notif_1 | SLF4J:
See http://www.slf4j.org/codes.html#StaticLoggerBinder
for further details.
notif_1 | == Spark has ignited ...
notif_1 | >> Listening on 0.0.0.0:5000
app_1 | App listening on port 8080
db_1 | 2015-06-21T22:18:26.845+0000 I NETWORK [initandlisten]
connection
accepted from 172.17.6.140:53805 #1 (1 connection now open)
app_1 | successful connection (first try)
calc_1 | Mortgage calculation service listening to
http://optimizer/api/Jobs/
optim_1 | Started
calc_1 | No more jobs !

```

Como muestra el comando `docker ps` (se debe lanzar desde otro terminal, el utilizado por el comando Docker Compose se reserva para mostrar los logs a medida que se ejecuta nuestra aplicación ejemplo), es la razón por la que no hemos usado la opción `-d`), los contenedores están correctamente ahí, con un nombre diferente a los utilizados en el enfoque manual y que utilizan sistemáticamente el prefijo `test`:

CONTAINER ID	IMAGE	COMMAND	
CREATED			
STATUS	PORTS	NAMES	
cc7655a998aa	mortgage_app:latest	"node index.js"	5
minutes ago			
Up 5 minutes	0.0.0.0:8080->8080/tcp	mortgage_app_1	
96b0e57806be	mortgage_notif:latest	"/usr/lib/jvm/java-8	5
minutes ago			
Up 5 minutes	5000/tcp	mortgage_notif_1	
8197ff3b16ba	mortgage_report:latest	"python -u webtest.p	5
minutes ago			
Up 5 minutes	5000/tcp	mortgage_report_1	
0fb0c17f3dab	mongo:3.0.3	"/entrypoint.sh mong	5
minutes ago			
Up 5 minutes	27017/tcp	mortgage_db_1	
c7b7b0d44815	mortgage_calc:latest	"dnx . run"	5
minutes ago			
Up 5 minutes		mortgage_calc_1	
665ea5a53961	mortgage_optim:latest	"/bin/sh -c 'sleep 1	5
minutes ago			
Up 5 minutes	80/tcp	mortgage_optim_1	

Para la nomenclatura, Docker Compose utiliza el nombre del proyecto (`mortgage`), seguido del nombre definido en el archivo `docker-compose.yml` para cada uno de los contenedores (por ejemplo, `app`, `notif` o `report`). El comando `docker images` permite ver que Docker Compose dispone de sus propias imágenes, construidas anteriormente por `docker-compose build`:

## c. Gestión de los contenedores

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
jpgouigoux/calculator 220MB	latest	7d3368ef373f	15 minutes ago
jpgouigoux/optimizer 80MB	latest	480e826c8f8c	16 minutes ago
jpgouigoux/reporting 733MB	latest	40c7f71a20ff	2 days ago
jpgouigoux/portal 84.9MB	latest	3947245ac2a3	2 days ago
jpgouigoux/notifier 840MB	latest	959ab45fa8e1	3 days ago
microsoft/aspnetcore 280MB	2.0	0bc7246be73f	4 days ago
node 65.3MB	8.7-alpine	a47a5669ac57	5 days ago
python 690MB	3.6	01fd71a97c19	7 days ago
mongo 301MB	3.2	16dc6ae8085e	7 days ago
microsoft/dotnet 1.64GB	2.0-sdk	2d1d3b4f6d02	2 weeks ago
microsoft/dotnet 219MB	2.0-runtime	4de086c811b1	2 weeks ago
java 643MB	8	d23bdf5b1b1b	9 months ago

Para abreviar, el comando `docker-compose` funciona de la misma manera que el comando `docker`, salvo que realiza la operación sobre todos los contenedores especificados en el archivo `docker-compose.yml` utilizado. De esta manera, `docker-compose stop` equivale a un `docker stop` sobre todos los contenedores, el `docker-compose up` anteriormente utilizado crearía los contenedores y los arrancaría, `docker-compose restart` permitiría volver a lanzarlos, el comando `docker-compose build` lanzaría la recompilación de todas las imágenes, etc. Al final, solo el comando `up` es un poco diferente del equivalente que es un `docker run` en modo unitario.

También existe `docker-compose logs`, pero hemos visto que los logs se mostraban automáticamente en modo dinámico cuando se utilizaba el comando `up`.

Con este propósito, después de la puesta en marcha del conjunto de contenedores, es posible detener todo mediante el acceso directo [Ctrl] **C**, que implica la parada limpia de los contenedores. La siguiente captura de pantalla textual muestra que un segundo uso del mismo acceso directo [Ctrl] **C** durante la parada limpia, forzará la parada inmediata de los contenedores, sin esperar a que se detengan en el tiempo asignado. A continuación se muestra una salida de los logs de un entorno en el que se ha

utilizado este método de forzado (vemos los dos textos "^C" en dos líneas de intervalo):

```
app_1      | POST /api/mortgages 200 133.961 ms - 148
notif_1    | Account used to send mail is jp.gouigoux@gmail.com
report_1   | Best repayment date is:
report_1   | 172.17.6.133 - - [21/Jun/2015 22:14:39] "POST
          /pdf?amountRepaid=1&replacementFixRate=1 HTTP/1.1" 200 -
          ^CGracefully stopping... (press Ctrl+C again to force)
          Stopping mortgage_app_1...
          ^CKilling mortgage_app_1...
          Killing mortgage_notif_1...
          Killing mortgage_report_1...
          Killing mortgage_db_1...
          Killing mortgage_calc_1...
          Killing mortgage_optim_1...
```

Si las descripciones de las imágenes han cambiado, el comando `docker-compose build` vuelve a compilar las imágenes y el comando `docker-compose up` volverá a crear los contenedores necesarios.

El comando `docker-compose ps` utiliza una visualización ligeramente diferente de `docker ps`, que tiene por objetivo listar todos los contenedores mientras que en nuestro caso, solo serán visibles los contenedores enlazados con el archivo de composición:

Name	Command	State	Ports
-----	-----	-----	-----
mortgage_app_1	node index.js	Up	0.0.0.0:8080->8080/tcp
mortgage_calc_1	dnx . run	Up	
mortgage_db_1	/entrypoint.sh mongod	Up	27017/tcp
mortgage_notif_1	/usr/lib/jvm/java-8-openjd ...	Up	5000/tcp
mortgage_optim_1	/bin/sh -c sleep 10000000 ...	Up	5004/tcp
mortgage_report_1	python -u webtest.py	Up	5000/tcp

Una vez que ha terminado el ciclo de vida de la aplicación, el comando `docker-compose rm -fv` eliminará todos los contenedores y los volúmenes asociados. Respecto al comando `docker-compose down`, permite una parada y después una eliminación de los contenedores, sin eliminación directa de los volúmenes y por lo tanto, será el comando preferente para detener un entorno aplicativo.

## 4. Paralelización de las operaciones

La separación clara entre la composición de una arquitectura y los comandos que permiten gestionarla, muestra mejor sus efectos

beneficiosos cuando nos interesamos por la escalabilidad de los servicios. Como se había explicado durante la descripción del funcionamiento de los servicios `optimizer` y `calculator`, es posible lanzar varias instancias de `calculator` para ir más rápido. En modo "manual", sería necesario añadir tantas líneas como fuera necesario, y esto en todos los scripts. Con Docker Compose, la implantación de tres contenedores de cálculo es mucho más sencilla:

```
docker-compose scale calc=3
```

El resultado de esta línea de comandos es la creación de tres instancias de contenedores de cálculo:

```
Creating mortgage_calc_1...
Creating mortgage_calc_2...
Creating mortgage_calc_3...
Starting mortgage_calc_1...
Starting mortgage_calc_2...
Starting mortgage_calc_3...
```

Si se vuelve a arrancar el entorno con `docker-compose up` y se genera un caso de prueba, la operación de todos los casos de cálculo de préstamo será mucho más rápida y los logs mostrarán claramente el hecho de que varios servicios compiten por tratar los cálculos de préstamo:

```
notif_1 | Account used to send mail is jp.gouigoux@gmail.com
optim_1 | 29 jobs created
calc_4  | 29 job(s) remaining
calc_5  | 29 job(s) remaining
calc_3  | 29 job(s) remaining
calc_4  | 26 job(s) remaining
calc_3  | 25 job(s) remaining
calc_5  | 14 job(s) remaining
calc_4  | 23 job(s) remaining
calc_3  | 23 job(s) remaining
calc_5  | 21 job(s) remaining
calc_4  | 20 job(s) remaining
calc_3  | 19 job(s) remaining
calc_5  | 19 job(s) remaining

[...]

calc_4  | 5 job(s) remaining
calc_3  | 4 job(s) remaining
calc_5  | 3 job(s) remaining
calc_3  | 3 job(s) remaining
calc_4  | 3 job(s) remaining
calc_5  | 1 job(s) remaining
calc_3  | 1 job(s) remaining
calc_4  | 1 job(s) remaining
calc_5  | No more jobs !
calc_3  | No more jobs !
calc_4  | No more jobs !
```

```
optim_1 | Index of best repayment month is 212  
report_1 | Best repayment date is: "2017-09-01T00:00:00"
```

Cuando la lista de jobs aparece en el servicio `optimizer`, los tres servicios de cálculo ven el mismo número de jobs que quedan por tratar, por ejemplo, 29. Pero terminado el primero, no ve más que 26, porque los otros dos también han trabajado durante este tiempo y han pasado la propiedad `Taken` a `true`, para indicar que el job se había tratado.

Las prioridades de las aplicaciones tienen un efecto extraño en la programación, pero todos los jobs se lanzarán y se tratarán. Esta escalabilidad se ha establecido de manera sencilla, gracias a la inversión de la dependencia entre `optimizer` y los servicios `calculator`. Como no es el primero que controla a los segundos (para - como se ha explicado más atrás - no formar un cuello de botella), es suficiente con añadir un servicio para que forme parte del cálculo.

Este punto no se había detallado, pero el código del servicio `optimizer` utiliza un mecanismo de bloqueo para que dos servicios de tipo `calculator` escriban al mismo tiempo en la lista de jobs. Esto no impide que un servicio se haga cargo de un job mientras que otro lo está modificando para indicar que lo ha cogido. Pero este lapso de tiempo de recuperación es tan bajo respecto al cálculo en sí mismo, que este caso es estadísticamente improbable. Incluso si en un caso entre mil, dos servicios calculan el mismo resultado de coste del préstamo, esto no sería gravísimo para el rendimiento. Desde un punto de vista funcional, esto será poco problemático, porque el servicio de cálculo es idempotente: para un mismo préstamo, que un servicio u otro, incluso varios, lo calculen, el resultado será necesariamente el mismo respecto al coste total del préstamo, porque los argumentos entrantes serán los mismos y el servicio no tiene estado. Hablamos de servicio sin estado (*stateless* en inglés), cuando el funcionamiento de una llamada no depende de las llamadas anteriores, en resumen, cuando el servicio no almacena ningún dato de una llamada a otra, garantizando de esta manera respuestas predecibles.

Esto no es más que un límite teórico del código en su versión 1.0, incluso si en la práctica el rendimiento se mejora de manera casi lineal.

## 5. Límites

Respecto a los límites, la aplicación que nos ha servido de ejemplo está lejos de ser un modelo de descomposición en micro-servicios. Para obtener alguna cosa fácilmente utilizable para nuestro contexto de aprendizaje, fue necesario recortar numerosas complejidades que no tenían sentido en un libro sobre Docker, pero que serían fundamentales en producción:

- El API sobre los jobs debe permitir listar únicamente los jobs en espera.
- También es necesario que este mismo API autorice solicitar el contenido de un único job por su identificador.
- La gestión optimista de asignación de jobs no es necesariamente algo malo, pero es necesario al menos comprobar el número de jobs tratados dos veces.
- Falta de asincronismo entre la redacción de un informe y su envío.
- Validación de las entradas en los API, en particular `BestRepaymentDate` que tiene una gestión del timeout un poco ligera, y pueda salir inmediatamente si los datos que se introducen no son válidos.
- Falta de funcionalidad de purgado de jobs.
- Etc.

# Explotación de una infraestructura Docker

## 1. La red en Docker

Sin darnos cuenta, hemos utilizado repetidamente funcionalidades de red de los contenedores, durante las aplicaciones prácticas anteriores. Es un signo positivo del hecho de que Docker haga tan transparente la gestión de la red, pero conviene profundizar ligeramente en el aspecto, para mostrar el resto de posibilidades. El objetivo de la presente sección no es entrar en todo los aspectos de la gestión red de Docker (para esto sería necesario un libro entero), sino explicar cómo funcionan las operaciones realizadas anteriormente y mostrar que la red se puede configurar de otra manera si es necesario.

### a. Modo de funcionamiento estándar (bridge)

Para simular una máquina estanca real, los contenedores tienen su propia interfaz de red (veremos algunas excepciones un poco más adelante). Por defecto, Docker funciona en modo `bridge` (*puen*te en inglés), es decir, que el demonio Docker establece una capa de indirección entre la interfaz de red de la máquina host y la creada para los contenedores. Esta interfaz de red intermedia se llama `docker0`.

El comando `ifconfig` ejecutado en una máquina en la que está instalado Docker, muestra las características de esta interfaz, además de las tradicionales interfaces de red ethernet `eth0` y bucle local `lo`:

```
jpg@Ubuntu1410:~$ ifconfig
docker0  Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
         inet addr:172.17.42.1  Bcast:0.0.0.0  Máscara:255.255.0.0
         adr inet6: fe80::5484:7aff:fefe:9799/64 Scope:Enlace
         UP BROADCAST MULTICAST  MTU:1500  Metric:1
         Paquetes recibidos:2378 errors:0:0 overruns:0 frame:0
         TX packets:3520 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 lg file transmission:0
         Bytes recibidos:150825 (150.8 KB) Bytes enviados:3449866
         (3.4 MB)

eth0     Link encap:Ethernet  HWaddr 08:00:27:8f:5b:2c
         inet addr:10.0.2.15  Bcast:10.0.2.255  Máscara:255.255.255.0
         adr inet6: fe80::a00:27ff:fe8f:5b2c/64 Scope:Enlace
```



```

UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
Paquetes recibidos:431534 errors:0:0 overruns:0 frame:0
TX packets:1264210 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 lg file transmission:1000
    Bytes recibidos:381926087 (381.9 MB) Bytes
enviados:2786326446
(2.7 GB)

lo        Link encap:Bucle local
          inet addr:127.0.0.1 Máscara:255.0.0.0
          adr inet6:::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          Paquetes recibidos:8475 errors:0:0 overruns:0 frame:0
          TX packets:8475 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 lg file transmission:0
          Bytes recibidos:953749 (953.7 KB) Bytes enviados:953749
(953.7 KB)

```

El mismo comando lanzado desde un contenedor creado puntualmente, justo cuando se lanza el mismo comando (gracias a la opción `--rm`, que elimina el contenedor en la salida), muestra la siguiente salida:

```

jpg@Ubuntu1410:~$ docker run --rm ubuntu:trusty ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:01
          inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:1/64 Scope:Link
          UP BROADCAST MTU:1500 Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:270 (270.0 B) TX bytes:90 (90.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr:::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

En el interior del contenedor se dispone de una interfaz de red, pero no es la misma que la disponible en la máquina host. La base de direcciones es la misma (172.17.0.0 con una máscara 255.255.0.0).

Una representación gráfica muestra mejor la apilación de las interfaces de red:




images/esquema25.png

La presencia de estas capas separadas permite realizar todas las operaciones de enrutamiento, filtro, etc. en función de las situaciones encontradas. Por defecto, Docker autoriza qué

paquetes salen de la red de la máquina host, y la capa `docker0` permite asignar una dirección IP a cada uno de los contenedores en una misma franja de direcciones. Esta es la capa intermedia que sirve de soporte a los enlaces entre contenedores, mostrados en el ejemplo más atrás en el presente capítulo.

El soporte de red tiene muchas opciones. Solo mostraremos tres, que normalmente son las que más se utilizan. En primer lugar, la opción `--add-host` permite añadir en la ejecución una entrada en el archivo `hosts` del contenedor y por tanto, enlazar un alias con la dirección de la máquina destino. Por ejemplo:

```
jpg@Ubuntu1410:~$ docker run --rm --add-host news:93.184.220.20
ubuntu:trusty ping news
PING news (93.184.220.20) 56(84) bytes of data.
64 bytes from news (93.184.220.20): icmp_seq=1 ttl=50 time=33.2 ms
64 bytes from news (93.184.220.20): icmp_seq=2 ttl=50 time=35.9 ms
^C
--- news ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 33.292/34.596/35.901/1.317 ms
```

 Preste atención a no abusar de esta opción, por ejemplo para establecer enlaces entre los contenedores. La opción `--link` es más potente, porque abstrae al usuario del conocimiento de la dirección IP asignada. Por lo tanto, solo conviene utilizar la opción `--add-host` para las máquinas externas.

A continuación, la opción `--dns` permite pasar la IP de un servidor DNS al contenedor.

Para terminar, la opción `--hostname` se utiliza para cambiar el nombre de máquina correspondiente al contenedor.

## **b. Modos de funcionamiento alternativos**

El modo `bridge` cubre casi todos los casos de uso estándares y es relativamente seguro, en el sentido de que si no utiliza la opción `-p` para asociar un puerto disponible por la máquina host a un puerto expuesto por el contenedor, estos permanecen sellados. Por lo tanto, es necesario exponer explícitamente un contenedor dentro del exterior, lo que es algo bueno.

Existen algunos casos particulares en los que la red se debe configurar de manera diferente, gracias a la opción `--net` en el comando `docker run`. Observe que no entraremos en la

complejidad de la configuración de red en el demonio Docker en sí mismo.

Un primer caso particular es hacer que un contenedor sea completamente estanco, es decir, que no exponga puertos para el consumo de información y que no pueda llamar más a la pila de red de la máquina host. Este modo de funcionamiento se asocia al valor `none` de la opción `--net`:

```
jpg@Ubuntu1410:~$ docker run --rm --net none ubuntu:trusty ping elpais.es  
ping: unknown host elpais.es
```

Un segundo caso particular es el inverso del primero y consiste en considerar el contenedor como un proceso no estanco y con los mismos permisos que los procesos locales de la máquina host respecto a la red: se podrá ver su dirección y será accesible desde el exterior (respetando las reglas del cortafuegos). Este modo de funcionamiento se asocia al valor `host` de la opción `--net`.

El diagrama de interfaz es el siguiente:

images/esquema26.png

Para terminar un tercer caso, que puede ser incluso más particular que los dos anteriores. Se corresponde con la puesta en común de la capa de red estanca entre dos contenedores, lo que se llama contenedores unidos. El diagrama es el siguiente:

images/esquema26.png

La implantación de este modo de funcionamiento es un poco diferente de los anteriores, porque no es suficiente con pasar un valor fijo a la opción `--net`: es necesario pasarle el nombre del primer contenedor con el que el segundo desea compartir la capa de red. Los siguientes comandos muestran un ejemplo con un contenedor que expone un servidor web y después, la comprobación de la actividad de red por medio de otro contenedor que tiene la misma pila de red (entre los dos comandos, arrancar un navegador en `http://localhost:8080` para crear la actividad):

```
jpg@Ubuntu1410:~$ docker run -d --name contenedor2 -p 8080:80 nginx  
d3c6353f7a4acf82c02dbc8d404c8b9ad4292d3ce89fb7fcfc9ece3ca67b8c0b
```

```
[desencadenar una actividad de red en el primer contenedor]

jpg@Ubuntu1410:~$ docker run --rm --name contenedor3 --net
container:contenedor2
ubuntu:trusty netstat -al
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:http                  *:.*                     LISTEN
tcp        0      0 d3c6353f7a4a:http     172.17.42.1:35156      ESTABLISHED

Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State      I-Node    Path
unix   3      [ ]        STREAM    CONNECTED  23165
unix   3      [ ]        STREAM    CONNECTED  23166
```

Los usos de esta opción están limitados a los contextos en los que un contenedor debe poder analizar la pila de red utilizada por otro. Podemos imaginar, por ejemplo, un contenedor dedicado a la supervisión de red o bien una Intrusion Detection System (un IDS, sistema de detección de intrusión de red, es una aplicación que busca trazas de comportamientos sospechosos en los accesos, para mejorar la seguridad de un sistema).

Docker todavía es joven y no es imposible que en los siguientes años, se establezcan buenas prácticas que aumenten el uso de estos casos particulares pero, por el momento, su uso es anecdótico.

### c. Soporte de los enlaces entre contenedores

El mecanismo de enlaces entre los contenedores se ha mostrado más atrás en el ejemplo de arquitectura de micro-servicios, en el que los enlaces permiten fácilmente relacionar cada uno de los servicios. La presencia de una sección dedicada a la gestión de red justifica volver a este punto particular, porque es la capa de red gestionada por Docker la que permite establecer los enlaces entre contenedores.

Cuando se crea un enlace de un contenedor a otro (observe que este enlace no es bidireccional), Docker recupera de manera dinámica la dirección IP del primero en su interfaz `docker0`, y después inyecta el enlace entre el nombre proporcionado en el enlace - que sirve de alias - y la dirección IP obtenida en el archivo `hosts` del segundo contenedor, arrancado con la opción de enlace. Se puede comprobar en los siguientes comandos la presencia de la entrada en el archivo de los alias en el segundo contenedor:

El alias `servidorweb` se ha relacionado con la IP `172.17.0.9`,

```

jpg@Ubuntu1410:~$ docker run -d --name contenedor2 -p 8080:80 nginx
677da274ee579202eab5efb0dbd2d94034fa2a2c61d9e74ab7500ffbe43e40c5
jpg@Ubuntu1410:~$ docker run --rm -it --link
contenedor2:servidorweb
ubuntu:trusty
root@6b566fceb2ae:/# cat /etc/hosts
172.17.0.11      6b566fceb2ae
127.0.0.1       localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0         ip6-localnet
ff00::0         ip6-mcastprefix
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
172.17.0.9      servidorweb 677da274ee57 contenedor2
root@6b566fceb2ae:/# exit
exit

```

que era la utilizada por contenedor2, lanzado sobre la imagen nginx.

Sobre todo, hemos hablado de la inyección en las entradas de alias para realizar el enlace, pero observe que Docker tiene en cuenta los procesos que no pueden utilizar la resolución de nombres y proporcionan variables de entorno, que devuelven la misma información:

```

jpg@Ubuntu1410:~$ docker run --rm -it --link contenedor2:servidorweb
ubuntu:trusty
root@1aa38d9da502:/# env
SERVIDORWEB_PORT_443_TCP=tcp://172.17.0.9:443
HOSTNAME=1aa38d9da502
TERM=xterm
SERVIDORWEB_PORT_443_TCP_PORT=443
SERVIDORWEB_PORT_443_TCP_PROTO=tcp
SERVIDORWEB_ENV_NGINX_VERSION=1.9.1-1~jessie


[...]

SERVIDORWEB_PORT_80_TCP=tcp://172.17.0.9:80
SERVIDORWEB_PORT_80_TCP_PROTO=tcp
SERVIDORWEB_PORT_443_TCP_DIR=172.17.0.9
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SERVIDORWEB_PORT_80_TCP_DIR=172.17.0.9
SHLVL=1
HOME=/root
SERVIDORWEB_PORT=tcp://172.17.0.9:80
LESSOPEN=| /usr/bin/lesspipe %s
SERVIDORWEB_NAME=/clever_darwin/servidorweb
LESSCLOSE=/usr/bin/lesspipe %s %s
SERVIDORWEB_PORT_80_TCP_PORT=80
_=/usr/bin/env

```

En los procesos de software, algunas veces es más sencillo acceder a una variable de entorno, por ejemplo si solo se trata de leer una información sin llamar al servicio en un primer momento. De manera general, las variables de entorno son el

medio más universalmente reconocido para intercambiar información sencilla entre dos contextos.

 La gestión de los enlaces permite a las personas encargadas del diseño de los contenedores, no preocuparse por la manera en la que se despliegan y se relacionan las unas con las otras. En este sentido, este mecanismo constituye una herramienta DevOps real.

#### **d. Otras opciones**

De nuevo dirigimos al lector a la documentación de Docker para ver la impresionante lista de las configuraciones de red, que responde a casi todas las necesidades, desde las más sencillas mostradas antes hasta las más sofisticadas, que no tienen lugar en un libro generalista.

La página <https://docs.docker.com/engine/reference/run/>, menciona todas las opciones de red accesibles para la ejecución de un contenedor. Respecto a la configuración a nivel del demonio Docker, se trata en la página <https://docs.docker.com/articles/networking/>.

Estos dos enlaces le mostrarán cómo gestionar las DNS, modificar la dirección MAC (*Media Access Control*) de una interfaz de red, cambiar la pasarela, controlar las direcciones IP de los contenedores, gestionar IPv6, etc.

#### **e. Límites de la capa de red existente**

A pesar de esta riqueza de opciones, Docker también tiene limitaciones en su capacidad de red. La más importante de ellas es que la interfaz de red común entre los contenedores, está relacionada a la máquina host. La consecuencia más visible es que, solo utilizando Docker es imposible realizar un enlace entre dos contenedores con dos demonios Docker funcionando en dos máquinas separadas.

Se han propuesto y desarrollado numerosas soluciones en los últimos años, pero un primer nivel de reducción de la oferta ya ha tenido lugar durante los años 2016 y 2017, sobre todo desde la versión 1.12 de Docker, que ha industrializado mucho el enfoque integrado de orquestador, llamado modo Swarm. Sin ser tan sofisticado como DC/OS de Mesos o Kubernetes de Google, el modo Swarm de Docker es un enfoque eficaz que permite

relacionar en red a los demonios Docker, de manera que aparezcan como un único demonio de gran capacidad, y además con una gestión de las redes overlay y un DNS automático que hace extremadamente sencillo su explotación conjunta, con el resto de herramientas del ecosistema Docker.

El funcionamiento de Swarm se mostrará en el siguiente capítulo, donde se implementará la aplicación ejemplo en un cluster de tres máquinas.

## 2. Los volúmenes Docker

### a. El problema de la persistencia

La gestión de las capas de Docker hace que las imágenes básicas para la puesta en marcha de una instancia de contenedor, jamás se modifiquen. Están en modo solo lectura y todas las modificaciones sobre el contenedor se realizan en una capa adicional, que almacena las modificaciones. A continuación, es posible utilizar la operación `commit` para almacenar estas modificaciones de manera definitiva en otra imagen.

Este método se adapta cuando el usuario está en fase de construcción de una imagen, pero no del todo para la explotación de una imagen existente.

Por ejemplo, imaginemos el caso de un contenedor que expone un servicio de base de datos. Por defecto, todas las modificaciones realizadas en la base de datos van a encontrarse en esta capa temporal en modo lectura/ escritura. ¿Qué sucedería si nadie piensa en utilizar la opción `commit` durante la parada del contenedor? Los datos de los usuarios recogidos pacientemente, que se tratan de comandos, peticiones de directorios en línea, etc., simplemente se perderán.

Evidentemente, este modo de funcionamiento es inaceptable y es por esto que Docker, ha previsto una gestión de los volúmenes para la persistencia de los datos.

### b. Los volúmenes como solución sencilla

Para explicar el principio de funcionamiento de los volúmenes, vamos a utilizar la imagen oficial de MongoDB. A continuación se muestra un extracto del Dockerfile para la imagen en versión 3.0:

La palabra clave `VOLUME` tiene el efecto de que los contenedores

```
FROM debian:wheezy

[...]

RUN set -x \
    && apt-get update \
    && apt-get install -y mongodb-org=$MONGO_VERSION \
    && rm -rf /var/lib/apt/lists/* \
    && rm -rf /var/lib/mongodb \
    && mv /etc/mongod.conf /etc/mongod.conf.orig

RUN mkdir -p /data/db && chown -R mongodb:mongodb /data/db
VOLUME /data/db

COPY docker-entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]

EXPOSE 27017
CMD ["mongod"]
```

instanciados en esta imagen, no llevarán las modificaciones realizadas localmente a /data/db (comportamiento estándar de MongoDB) en la capa en modo escritura, sino a un directorio gestionado por Docker y directamente asociado al disco duro de la máquina host.


Vamos a mostrar esto con una operación corta:

Lance un contenedor basado en la imagen `mongo`, utilizando la línea de comandos siguiente:

```
docker run -d --name db -p 27017:27017 mongo:3.0.3
```

Después conéctese a la base de datos MongoDB (el puerto 27017 es el utilizado por defecto):

```
mongo
```

 Si no dispone del cliente `mongo` para acceder a la base MongoDB, puede instalarlo con el comando `apt-get install mongodb-clients` en Ubuntu o el equivalente, en su distribución.

Posiciónese en la base `mydb` con el comando `use mydb`.

Realice una modificación en esta base, por ejemplo añadiendo una entrada en una colección `personas`, como sigue:

```
db.personas.insert({ nombre: "Lagaffe", apellidos: "Gaston" })
```

Salga de la base de datos con el comando `exit`.

Lance un comando `docker inspect db` para ver los



argumentos del contenedor db y busque una información que contenga el nombre "volume", como sigue:

```
"Volumes": {
  "/data/db":
"/var/lib/docker/vfs/dir/1e3960af56b96bda0d3e808d6321cee6a712bbf1
7b0f9d84e664dd8da9401b5c"
},
"VolumesRW": {
  "/data/db": true
}
```

/var/lib/docker/vfs/dir/1e3960af56b96bda0d3e808d6321cee6a712bbf17b0f9d84e664dd8da9401b5c es el directorio en el que Docker ha redirigido el contenido escrito por el servidor MongoDB en /data/db, durante la ejecución del contenedor db. Vamos a lanzar un segundo contenedor que apunta a este volumen, para mostrar el contenido del disco duro que aloja los datos de la base de datos:

Arranque un contenedor que relacione explícitamente este volumen con /data/db, utilizando otro puerto para no tener conflictos con 27017, así como otro nombre, ya que por el momento db está siendo utilizado:

```
docker run -d --name dbalt -p 27018:27017 -v
"/var/lib/docker/vfs/dir/1e3960af56b96bda0d3e808d6321cee6a712bbf1
7b0f9d84e664dd8da9401b5c":/data/db mongo:3.0.3
```

Conéctese a la base MongoDB expuesta por este nuevo contenedor:

```
mongo -p 27018
```

Posiciónese a la base mydb.

Busque los datos con el comando `db.personas.find()`.

Debería obtener una visualización como la siguiente, lo que prueba que hemos recuperado estos datos desde el volumen:

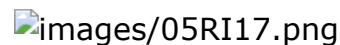
```
> db.personas.find()
{ "_id": ObjectId("559001f2ffff688b1a73b7075"), "nombre":
"Lagaffe", "apellido": "Gaston" }
```

Este ejemplo muestra que simplemente usando la palabra clave VOLUME en el archivo Dockerfile, con el nombre del directorio donde guardar, permite salvar el problema enunciado al inicio de esta sección. Ahora vamos a explorar algunos usos más concretos de los volúmenes.

### c. Enlace directo sobre un directorio local


El primer uso sofisticado de los volúmenes consiste en no dejar que Docker elija un directorio, sino en pasarle como argumento aquel que queremos que sea el directorio donde se apunten los datos asociados al volumen. Por ejemplo, podemos lanzar otro contenedor `mongo` en el que indicamos que el volumen asociado a `/data/db` debe apuntar al directorio de persistencia especialmente creado para este efecto en la máquina host.

Explorando este directorio, veremos la arborescencia típica creada por el servidor MongoDB para almacenar los datos que se le confían:



Este enfoque es un poco más complejo de inicializar (hay una opción más en el comando `docker run`), pero es más fácil de explotar (los datos están donde los hemos pedido y no en un directorio enterrado en las entrañas de Docker, y que hace necesario inspeccionar el contenedor implicado para localizarlo).

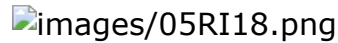
Por el contrario, la asignación automática del volumen plantea la ventaja de que no es necesario para la explotación anticipar la manera en la que el almacenamiento se genera.

 Como para los volúmenes asignados automáticamente, es posible pasar varios argumentos de definición de volúmenes.

### d. Compartir volúmenes

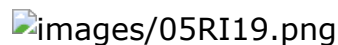
De la misma manera que los enlaces entre contenedores mostrados más atrás permiten conectar un alias de un contenedor a la IP de otro utilizando una sencilla opción, es posible compartir volúmenes entre dos contenedores sin tener que encontrar el identificador del volumen con un `docker inspect`.

La opción a utilizar es `--volumes-from`, seguida del nombre del contenedor destino. Por ejemplo, partiendo del ejemplo anterior, es posible conectar un segundo contenedor sobre el volumen expuesto por el primero, lo que permitirá un acceso idéntico al contenido de `/data/db`.



En caso de compartir el directorio, es necesario comprobar que los dos contenedores gestionan correctamente los permisos de precedencia en la escritura de los archivos. En el ejemplo anterior, se recomienda que el segundo contenedor no toque el contenido del directorio MongoDB, porque se corre el riesgo de corromper los datos.

Con el objetivo de dar seguridad, puede ser útil recurrir a la opción `:ro`, que permite crear un volumen en modo solo lectura, y que también funciona con `--volumes-from`, como muestra la siguiente imagen de nuestro ejemplo:



Esta funcionalidad normalmente se utiliza en el marco de las infraestructuras Docker, para permitir una copia de seguridad sencilla de los contenidos del volumen: a intervalos regulares, un contenedor se lanza con un enlace en modo solo lectura sobre los volúmenes de los contenedores a copiar, se genera un archivo a partir de este contenido (con ayuda de la herramienta `tar`, por ejemplo) y se envía a un NAS (*Network Attached Storage*), en sí mismo unido por otro enlace de volumen. Entonces, el contenedor se detiene y purga y la próxima operación de copia de seguridad utilizará uno nuevo, lo que permite limitar los efectos.

En el caso de que un servidor de aplicaciones bloquee la lectura de los archivos, es necesario proceder de otra manera y utilizar la función dedicada del servidor, para la exportación de un archivo. La práctica estándar es lanzar el comando en el contenedor con un comando `docker exec`.


## **e. Gestión de los volúmenes huérfanos**

La securización de los datos garantizada por Docker gracias a los volúmenes, evidentemente tiene un precio, ya que existe la posibilidad de generar volúmenes huérfanos si los contenedores no se detienen de manera limpia. Si Docker encuentra el comando `VOLUME` en el `Dockerfile`, el directorio correspondiente se almacenará en el disco duro, incluso si la persona que lanza el contenedor no ha enviado explícitamente una opción en este sentido y por lo tanto, puede que no sea consciente.

Mientras se respeten las buenas prácticas y los contenedores se eliminen con la opción `-v`, no hay problema: Docker contará el número de contenedores que acceden al volumen, incrementándolo en cada enlace por un contenedor y disminuyéndolo para cada contenedor correctamente eliminado. Para terminar, cuando el número de contenedores asociados al volumen sea cero, se purgará el volumen.

Por el contrario, si solo se detiene un único volumen con el sencillo comando `docker rm` sin la opción de gestión del volumen, el contador nunca bajará a cero. Por lo tanto no se eliminará, sino que ningún contenedor utilizara más su contenido. En este caso el volumen se llama huérfano.

El problema asociado a los volúmenes huérfanos es que utilizan espacio en disco para nada. Afortunadamente, existen scripts para eliminarlos, por ejemplo el disponible en la página <https://gist.github.com/eliasp/7720943>. Preste atención con no utilizar este script erróneamente. En particular, jamás se debería haber implementado en una plataforma de producción.

 Los riesgos de pérdida de datos como consecuencia de una utilización incorrecta, son reales.

## **f. Sofisticación del enfoque**

Todo el enfoque alrededor de los volúmenes, se debe adaptar al entorno. Los ejemplos anteriores son sencillos por diseño, porque se trata de demostrar la funcionalidad en lugar de perder al lector en las gestiones de permisos, la compartición de directorios, etc. Sin embargo:

- Un uso más realista para la copia de seguridad apuntará a un NAS o una bahía SAN (*Storage Area Network*).
- La copia de seguridad se asociará a una tarea planificada supervisada y probada por un mecanismo independiente.
- La persistencia sería potencialmente generada por un cluster dedicado, con un enfoque como CouchDB o MongoDB en versión cluster.
- Se debería implementar un sistema de permisos para que no importe qué contenedor acceda a los datos de otro.
- Etc.

## g. Aplicación para la gestión de los logs

El enfoque "estándar" de la gestión de los logs en Docker, consiste en sacar todas las trazas por los contenedores en la salida estándar de la consola, de manera que se pueda explotar con el comando `docker logs`. Este enfoque tiene la ventaja de ser muy sencillo.

Sin embargo, si hay procesos aplicativos que utilizan un directorio para almacenar los logs, la gestión por un volumen permitirá centralizar esto en un directorio de la máquina host, haciendo que parezca que el contenedor tiene un directorio fijo. De esta manera, el siguiente comando tiene la ventaja de que el código de la aplicación asociada simplemente podrá escribir sus archivos en `/log`, sin preocuparse de tener que cambiar de directorio un día:

```
docker run -v /tmp/trazas:/log apliservidor
```

Desde el punto de vista del código de generación de las trazas, el trabajo se simplifica: es suficiente con escribir el archivo en `/log` y por supuesto, este directorio se puede escribir en duro en el código, porque el mecanismo de contenedor permitirá si es necesario mostrarlo en otro lugar. Esto simplifica mucho la gestión de los logs en el código de un servidor de software.

Otro servidor podría abrir su código que envía por defecto los logs a `/tmp`, lo que no plantearía ningún problema para centralizar en `/tmp/trazas`:

```
docker run -v /tmp/trazas:/tmp segundaapli
```

Al final, todos los logs serán accesibles en un directorio central de la máquina host (que puede apuntar a un lugar compartido externo), lo que facilitará el análisis de la arquitectura de micro-servicios. Es necesario respetar una convención sobre la nomenclatura de los archivos, de manera que los servicios no generen archivos con el mismo nombre. En general, esto no plantea problemas, los archivos de logs son cíclicos y utilizan nombres basados en la hora concreta de implementación u otro criterio, que permita diferenciarlos claramente. Una ventaja de esta centralización es que el almacenamiento y la purga de los logs se centralizan, lo que hace mucho más sencilla la implementación de una política de compartición para su gestión.

Para terminar, en los casos más industriales con numerosos servicios, será necesario ir un poco más allá y prever la

utilización de un syslog para recoger los logs, incluso pasar por una pila ELK (*ElasticSearch*, *LogStash*, *Kibana*) para su explotación.

## **h. Ir más lejos con los volúmenes**

La puesta en red del demonio Docker con ayuda del modo Swarm modifica sensiblemente la gestión de los volúmenes, pero el objetivo del presente libro es mostrar los principios y la aplicación de un Docker único.

## 6. DESPLIEGUE EN UN CLUSTER

### Descripción global del enfoque

#### 1. Objetivo

Hasta ahora, todos los despliegues se han realizado utilizando la máquina local como host de los contenedores. Cuando hablamos de despliegue, el registro permite poner a disposición de máquinas exteriores las imágenes creadas en una máquina dada, pero no impide que estas máquinas exteriores después de conectarse al registro para recuperar la imagen, instancien los contenedores correspondientes de manera local. El funcionamiento sigue siendo unitario y finalmente, no muy diferente de un despliegue sobre la máquina que ha servido para crear la imagen.

En el presente capítulo, vamos a estudiar los medios de desacoplar esta lógica y separar la máquina que controla el despliegue de la o las máquinas que van a llevar de manera efectiva los contenedores. El objetivo principal de este desacoplamiento es poder llegar fácilmente a varias máquinas para la segunda parte, y de esta manera poder extender fácilmente su capacidad de despliegue de contenedores, añadiendo recursos de hardware de manera transparente respecto al despliegue. En resumen, el objetivo del clustering de los contenedores es permitir el despliegue de contenedores sobre un conjunto de máquinas que se comportan como una única y enorme entidad host de contenedores.

Otro objetivo que está muy relacionado, es desarrollar un número variable de instancias de algunos contenedores, de manera que se puede equilibrar mejor la carga de estos contenedores. Después de todo, es una de las razones principales por las que habíamos descompuesto una aplicación en micro-servicios: para poder fácilmente jugar con el número de instancias y los recursos asignados a cada servicio, para alcanzar un equilibrio entre estos servicios, sinónimo de utilización óptima de los recursos disponibles.

#### 2. Situación actual

Hoy en día hay varios orquestadores muy completos que soportan contenedores Docker, entre los que encontramos DC/OS de Mesosphere y Kubernetes de Google. Este último ocupa el 40% del mercado de los orquestadores en el momento de escribirse estas líneas, y cada vez más se sitúa en una posición de liderazgo. Permite operaciones muy complejas y despliegues muy largos. Sin embargo, su complejidad le pone fuera de juego para despliegues sencillos. En nuestro caso de ejemplo en particular, vamos a utilizar el modo Swarm de Docker, que se implementará fácilmente.

El modo Swarm de Docker es un sistema de clustering complejo y orquestación de los contenedores implementado nativamente en el producto desde su versión 1.12. Con anterioridad, existía un software externo con el nombre de Docker Swarm. Hoy en día el nombre es ligeramente diferente: Swarm para Docker. A continuación utilizaremos simplemente la expresión Swarm.


Como se ha explicado más atrás, la ventaja de Swarm sobre otros orquestadores es que es muy sencillo que una persona acostumbrada a utilizar el cliente Docker, tome el control de la herramienta, que reutiliza de la mejor manera posible los API de Docker, soportando el uso de Docker Compose, lo que permite retomar una gran parte de las costumbres de trabajo desarrolladas en los capítulos anteriores.

### 3. Desarrollo del ejemplo

Como en el capítulo sobre la implementación de una arquitectura de micro-servicios con Docker, vamos a proceder por etapas y mostrar, con un enfoque práctico, todas las operaciones necesarias para la implantación de esta misma aplicación en un cluster de máquinas.

Empezaremos por reservar máquinas en el cloud, que pondremos en la red Docker usando comandos Swarm adaptados. A continuación, tomamos el control sobre este cluster con un cliente remoto. Después de esto, el archivo `docker-compose.yml`, mostrado en el capítulo anterior, se modificará para adaptar su comportamiento al despliegue en red. Para terminar, cuando la aplicación de ejemplo sea funcional, mostraremos cómo escalar el servicio de cálculo para acelerar la operación de cuello de botella de la aplicación ejemplo, a saber, el cálculo del coste total de un préstamo en función de su fecha de reembolso.



 Las máquinas utilizadas para el cluster serán instancias CoreOS, arrancadas en Azure pero el lector es libre de crear cualquier conjunto de máquinas para las pruebas. La única condición es que estén relacionadas por una red, de manera que funcionen en conjunto. Estas máquinas pueden ser físicas, virtuales, locales, remotas o en cloud. También se pueden reservar manualmente, por Vagrant, Docker Machine, suscripción a cualquier cloud. El único requisito aplicativo previo es que dispongan de una versión reciente de Docker.

## 4. Advertencia

El objetivo aquí es mostrar la puesta en red de una aplicación, pero no entramos en los detalles de la implementación de un cluster Swarm o del mantenimiento en condiciones operativas de una aplicación Docker Compose, no más de los detalles que mostraremos sobre cómo Docker Machine puede mejorar la preparación del cluster.

# Montaje de un cluster Swarm

## 1. Funcionamiento resumido de Swarm

Aunque se trate de una herramienta ya muy sofisticada sobre la que podríamos escribir centenares de páginas, se puede implementar un cluster Swarm y explotar con muy pocos conocimientos, que vamos a resumir aquí.

En primer lugar, un cluster Swarm divide las máquinas que gestiona (se habla de nodos) en dos categorías, a saber, los agentes y los managers.

Los agentes son nodos de pura ejecución: su trabajo consiste en encargarse de los contenedores, pero ellos no deciden qué contenedores ni las condiciones de puesta en marcha o escalabilidad.

Los nodos administradores son los que centralizan esta orquestación y justamente van a decidir qué contenedor se arranca en qué demonio, con qué método de reparto de trabajo, etc. En resumen, son la parte inteligente de la red. Sin embargo, por defecto, un nodo administrador también es un agente, es decir puede hacer funcionar los contenedores como el resto. Esta elección tiene sentido visto que la carga de orquestación es tan baja y que decidir un nodo solo para esta tarea, sería un desperdicio de recursos.

Cuando se inicializa un cluster Swarm en una máquina (con un comando que se mostrará más adelante), el demonio se convierte automáticamente en un manager, y proporciona dos tokens que permiten, siempre que la máquina tenga acceso a la red de la primera, transformar el demonio de una segunda máquina en un manager asociado, o bien en un agente al servicio del primer manager. En nuestro ejemplo, crearemos un cluster muy sencillo con un manager y dos agentes.

Una vez que el cluster se haya implementado, aparece el conjunto de los demonios desde el punto de vista exterior, como un único demonio con una capacidad más grande. Para mostrar esto, una vez que se haya formado el cluster, nos conectaremos con una tercera máquina, en este caso una máquina Ubuntu que tendrá el papel de administración del cluster. Esta máquina solo necesitará el cliente Docker para realizar su trabajo de control (hablamos de

Docker CLI, para *Command Line Interface*, interfaz en línea de comandos).

Esto resume los conocimientos básicos del mecanismo Swarm. El resto consiste en realizar los primeros pasos de la puesta en práctica.

## 2. Descripción de las máquinas utilizadas

Como se ha indicado anteriormente, las tres máquinas implantadas para este ejercicio, son instancias Azure de tipo CoreOS:



El simple hecho de crearlas en el mismo espacio de recursos que comparte la red, hará que se conecten por una subred dedicada, ajustando el problema de su accesibilidad la una a la otra. La primera de las instancias implantadas se abrirá en Internet, asignándole un nombre DNS:




La topología obtenida será parecida a la siguiente:



Con el objetivo de hacer que nuestro cluster de demonios sea accesible desde una máquina completamente exterior (en este caso una máquina de administración local del autor), el nodo Network Security Group de la primera máquina servirá para abrir temporalmente el puerto 2375:



 Se entiende que, en un modo de producción industrial, se reforzaría la securización de este intercambio, pero este no es el tema del presente libro. Aconsejamos al lector la lectura de la explicación de los comandos en <https://docs.docker.com/engine/security/https/>, para realizar esta operación, que es imprescindible (como el autor ha podido comprobarlo durante la creación de los ejemplos, un puerto 2375 es "visitado" muy rápidamente y al cabo de algunas horas, se han tenido que eliminar procesos piratas de extracción de


moneda virtual, apenas abierta la presencia en Internet). La implementación de una conexión segura por medio de un certificado lleva su tiempo, necesita algunas competencias de administrador de sistemas, pero es absoluta e indiscutiblemente obligatorio para cualquier otro entorno de pruebas rápidas y de ejercicios de aprendizaje de la tecnología. Para los usos menos importantes, también es posible simplemente restringir el acceso al puerto 2375 a algunas IP sobre las que el administrador tiene el control. El resultado es un poco menos práctico, porque las IP pueden cambiar, pero esto permite un primer nivel de seguridad no despreciable y en la mayor parte de los casos, es suficiente.

También se deberá abrir el puerto 8080, porque es el que utilizaremos para exponer el servicio lanzado en Swarm.

De nuevo una vez más, la elección del modo de preparación de las máquinas destino del cluster Swarm es eminentemente personal, pero sería necesario hacer una elección entre las múltiples posibilidades. En particular, sería posible utilizar ofertas llamadas Container As A Service, como Azure Container Service, que proporcionan directamente un cluster con un número de máquinas especificado, incluso la elección del orquestador preconfigurado. Esto sería claramente más rápido, así como menos pedagógico.

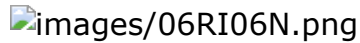
A título informativo, a continuación se muestra la interfaz de creación de un servicio Azure Container Service, en la que se ve la elección de los orquestadores disponibles durante la captura de pantalla:

images/10.png

 Es posible obtener una cuenta de prueba en Azure con un crédito de 150 horas gratuitas, lo que es más que suficiente para realizar los siguientes ejercicios. La página a visitar es <https://azure.microsoft.com/es-es/pricing/free-trial/>.

### 3. Inicialización del cluster

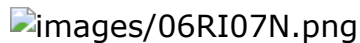
Para inicializar el cluster, es suficiente con conectarse a una de las máquinas que se convertirán en el primer nodo de tipo `manager`, y lanzar el comando `docker swarm init`. Este comando genera un resultado parecido al siguiente:



Además del retorno del comando, Docker nos indica cómo hay que continuar, a saber, lanzar el comando proporcionado en el resto de nodos que se van a unir al cluster Swarm. Esto es lo que vamos a hacer en la siguiente sección.

## 4. Adjuntar el resto de máquinas

Para adjuntar el resto de máquinas, también nos conectamos a estas máquinas y esta vez, el comando a lanzar es el proporcionado más atrás. Por ejemplo:



Esta vez, el retorno del comando se reduce a la información según la cual, el nodo se ha unido correctamente al cluster Swarm. El comando se reproduce en la tercera máquina y nos podemos desconectar de estas tres máquinas, porque el comando `docker swarm init` y los dos `dockers swarm join` serán los únicos realizados sobre el cluster, en sí mismo. Ahora vamos a tomar el control del cluster desde una máquina completamente exterior, para simular de manera más realista el trabajo de un administrador "de aplicaciones". El administrador "de sistemas" ha terminado su trabajo de preparación del cluster, de apertura de puertos de red, etc.

## 5. Control remoto

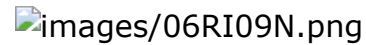
El resto de operaciones que siguen se realizarán desde una máquina local Ubuntu, pero para esto es necesario indicar que el demonio que hay que controlar es el del cluster. El comando a utilizar es el siguiente (con el nombre DNS de nuestro ejemplo a modificar, por supuesto):

```
export DOCKER_HOST=swarmjpg.westeurope.cloudapp.azure.com:2375
```

A continuación, un primer comando a utilizar antes que cualquier otra cosa es `docker info`, que va a permitir validar que Docker funciona bien en modo Swarm y que las tres máquinas previstas, están presentes y son accesibles.



Otra manera de comprobar la correcta estructura del cluster, es llamar al comando `docker node ls` sobre uno de los managers (la máquina `node1` a la que estamos conectados desde el puesto Ubuntu, en nuestro ejemplo):



El comando nos indica el número de nodos con su identificador y su nombre de host. Después indica si los nodos están disponibles y activos y su cualificación como manager. Siempre hay un manager que es líder, pero puede haber otros listos para tomar el relevo si es necesario. Un nodo de tipo agente puede ser "promovido" al rango de manager si es necesario, para garantizar esta robustez. Para terminar este rápido recorrido, el asterisco indica el nodo desde el que se envía el comando.

El cluster Swarm está preparado y es funcional. Por lo tanto, estamos preparados para pasar a la próxima etapa, que es desarrollar nuestra aplicación ejemplo.

# Despliegue en el cluster Swarm

## 1. Aspectos principales de los servicios

Para realizar este despliegue, en primer lugar va a ser necesario explicar rápidamente los conceptos de servicios, tareas y otras nociones relacionadas con la utilización de Swarm. De nuevo, el objetivo es dar un barniz mínimo para que el lector de este libro tenga unas nociones sobre conceptos. El siguiente diagrama explica con una imagen y pocas frases, la manera en la que Swarm permite el despliegue de aplicaciones con ayuda de "servicios" (en el sentido en que se han definido en el archivo Docker Compose).



Para simplificar el enlace con los comandos Docker, utilizaremos mayoritariamente las palabras en inglés para describir todas las relaciones del esquema anterior:

- Un stack S agrupa un conjunto de servicios A, B y C.
- Este stack S es el que se va a desplegar en un Swarm W.
- El Swarm W es un conjunto de demonios D1, D2 y D3 que funcionan en conjunto, por medio de una primera capa de red overlay (no representadas en el esquema por razones de claridad).
- Cada servicio se relaciona con una imagen y estas imágenes A, B y C se pueden encontrar en un registro R. Las restricciones y advertencias se han expuesto más atrás en el caso de una imagen construida durante la creación del servicio. Veremos un poco más adelante, que el registro se convierte en obligatorio en el caso del despliegue de un stack.
- Los servicios también definen un escalado del despliegue, por ejemplo tres instancias para el servicio B. Por defecto, el resto de servicios solo tendrán una única instancia.
- Estas instancias se llaman tareas y es el orquestador Docker Swarm el que se encarga de desarrollar la mayor parte de las tareas sobre el cluster que le piden los diferentes servicios del stack S.

- Las tareas A1, B1, B2, B3 y C1 se reparten en los diferentes nodos del cluster por el orquestador, que va a decidir el host (D1, D2 o D3) sobre el que se crearán, la manera en la que se reparten e incluso volver a arrancarlos en caso que sea necesario.
- Una tarea se ejecuta lanzando un contenedor Docker, basado en la imagen correspondiente del servicio asociado, que va a buscar en el registro y almacena en la caché local.
- Todas estas tareas se ejecutan dentro de una segunda red overlay, llamada N, que se ha creado superponiendo la red que agrupa las máquinas del Swarm entre ellas, para separar las instancias de servicios de este stack de las de otro stack.
- Si se ejecuta otra tarea en un mismo demonio Docker, como en el caso del demonio D2 en nuestro diagrama, será estanca respecto al stack S.
- De la misma manera, un contenedor lanzado en local en el demonio D1 (incluso si esto no sucede normalmente), no tendrá acceso al resto de contenedores o tareas que funcionan en este mismo demonio, con mayor motivo sobre las lanzadas en el resto de demonios.

Este diagrama debería haber aclarado los aspectos de vocabulario, pero solo un ejemplo permite validar el funcionamiento tal y como se describe. Sería posible desarrollar los servicios uno tras otro para mostrar en profundidad el funcionamiento en conjunto de estos conceptos Swarm, pero nuestro objetivo en este libro es centrarnos en el funcionamiento de la aplicación ejemplo. Se utilizará el comando más sencillo.

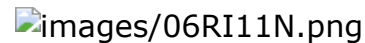
## 2. Envío de las imágenes al registro

Antes de lanzar el comando propiamente dicho, es necesaria una operación. En efecto, para que el stack funcione en el cluster Swarm que no conoce a priori, las imágenes que hemos creado en otro entorno, es necesario que los nodos puedan recuperarlas en un registro. En el ejemplo siguiente, las imágenes se publicarán en DockerHub con los nombres completos elegidos anteriormente.


Docker Compose dispone de un comando integrado que permite enviar todas las imágenes de golpe (salvo las imágenes oficiales, por supuesto) al registro público, a saber `docker-compose push`.

La ejecución del comando provoca enviar todas las imágenes a DockerHub, desde la máquina:





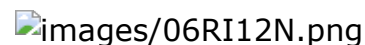
Atención, el comando `docker-compose push` solo hace esto: mandar las imágenes al registro. En particular, no verifica que las imágenes estén compiladas recientemente, no las vuelve a compilar ni las verifica antes del envío. Como regla general, la máquina que ha servido para poner a punto la aplicación, necesariamente tendrá las imágenes más actualizadas en su caché de registro local, cuando se manipulan varias máquinas en desarrollo, rápidamente se deja de tener todo actualizado. Por lo tanto, un comando `docker-compose build` no hará ningún mal antes de desencadenar el `push`, especialmente en la orquestación de la caché Docker que ejecuta este primer comando casi instantáneamente si no ha habido modificaciones en el contenido.

 Esta operación es una buena razón para guardar los atributos `build` en el archivo `docker-compose.yml`, incluso si no son útiles para el despliegue del stack como vamos a ver a continuación.

### 3. Ejecución de un stack

Una vez definidas las imágenes, la primera etapa consiste en crear un stack, es decir, un conjunto de servicios. Como el archivo `docker-compose.yml` que habíamos utilizado en el capítulo anterior estaba en versión 2, definía ya los servicios; por lo tanto, se va a usar el método de creación de un stack que se basa en este tipo de archivos, porque permite desarrollar la aplicación en el Swarm en una única línea de comandos, a saber `docker stack deploy`, la opción `-c` que permite pasar como argumento el archivo `docker-compose.yml`.


Una utilización directa del archivo anterior devuelve un error:



El mensaje es explícito: el despliegue de un stack por este modo hace necesario que el archivo Docker Compose esté en versión 3. Es la única modificación que hay que hacer en el archivo anterior, para llegar a esto:

Observe que los enlaces ya se habían eliminado del archivo,

```
version: '3'
services:
  mongo:
    image: "mongo:3.2"
  optimizer:
    build: ./optimizer
    image: jpgougoux/optimizer
  calc:
    build: ./calculator
    image: jpgougoux/calculator
    environment:
      - "URLOptimizerJobs=http://optimizer/api/Jobs/"
  reporting:
    build: ./reporting
    image: jpgougoux/reporting
  notifier:
    build: ./notifier
    image: jpgougoux/notifier
    environment:
      - SMTP_AUTH_LOGIN=jp.gougoux@gmail.com
      - SMTP_AUTH_PASSWORD=${GMAIL_PWD}
  app:
    build: ./portal
    image: jpgougoux/portal
    ports:
      - "8080:8080"
```

 porque se identificaron como no útiles en nuestro caso, al final del anterior capítulo.

El archivo se puede modificar para incluir su contraseña de mensajería, pero también es posible pasar a la ejecución del comando, de la misma manera que el comando `export` se utiliza para apuntar al cliente Docker sobre el cluster Swarm. Si todo funciona bien, el comando le enviará el siguiente mensaje:

 images/06RI13N.png


La advertencia relacionada con la primera línea (`Ignoring unsupported options: build`), se explica por el hecho de que un Docker Compose en versión 3, sirve al despliegue en producción sobre un cluster y por lo tanto, no se usa para crear las imágenes. En efecto, no tiene ningún sentido difundir el código fuente en el cluster, que solo conoce las imágenes compiladas. Esta es la misma razón por la que las opciones de tipo links ya no están de actualidad en la versión 3 de la gramática Docker Compose: como es el orquestador el que va a gestionar el reparto de los contenedores en los diferentes demonios, también va a gestionar el hecho de que estos contenedores puedan hablarse en una red dedicada al stack. La pregunta de los enlaces no se plantea, porque Swarm hace que cada contenedor del stack pueda

hablar a sus "hermanos", llamándolos por el nombre del servicio, garantizando una estanqueidad respecto al resto de contenedores correspondientes a otras aplicaciones.

Observe que si obtiene un mensaje como el siguiente, el problema viene del hecho de que no apunta al cluster, sino al demonio local (atención, el comando `export` no es persistente y vuelve a lanzar un terminal, será necesario de nuevo asignar el valor de `DOCKER_HOST`):

```
This node is not a swarm manager. Use "docker swarm init" or  
"docker swarm join" to connect this node to swarm and try again.
```

Algunos minutos más tarde (el tiempo depende mucho de las velocidades de descarga de las imágenes), la aplicación está preparada para su uso. Aunque esto no se hace en un entorno industrial, donde cualquier acción pasa por las máquinas de administración, vamos a echar un vistazo a las máquinas que componen el cluster por razones pedagógicas. La máquina `node1` sirve para dos contenedores:

images/06RI14N.png

Es igual para los otros dos nodos de nuestro cluster:

images/06RI15N.png

images/06RI16N.png

Swarm ha elegido un reparto por defecto de los contenedores, no tienen ninguna información sobre su necesidad de recursos y por lo tanto, ha distribuido linealmente los seis contenedores de aplicación de nuestro ejemplo, en los tres nodos. No hay preferencia particular sobre el reparto y, según su contexto, podría ser diferente.

En todos los casos se lanza la aplicación. El portal se puede utilizar con un navegador web como antes:

images/67.png

## 4. Escalado

Como en el modo de funcionamiento con un solo demonio, el comando `scale` de Docker Compose permite aumentar el número de contenedores de cálculo. Sin embargo, esta vez el servicio sigue siendo único y es el número de tareas lo que aumenta. Por supuesto, al final los contenedores son más numerosos, pero todo esto se hace de una manera mucho más controlada por Swarm. En particular, si hay un puerto expuesto, Swarm se ocupa de realizar un reparto de la carga en los diferentes contenedores que implementan el servicio.

En el caso particular del servicio de cálculo, resulta que no se utiliza ningún puerto (como hemos explicado, por razones de mayor escalabilidad es lo contrario, es decir, que cada uno de los contenedores gestione los jobs extraídos desde el contenedor `optimizer` que los expone).

El comando que se debe utilizar para pasar a cinco el número de tareas, es el siguiente:

```
docker service scale mortgagestack_calc=5
```


El nombre exacto del servicio se pueda encontrar con el comando `docker service ls`. Siempre se usa como prefijo el identificador utilizado por el stack. En nuestro caso, este comando provoca la siguiente visualización:


ID	NAME	MODE
REPLICAS		
IMAGE	PORTS	
w4fka8xw9w23	mortgagestack_app	replicated 1/1
jpgougoux/portal	*:8080->8080/tcp	
07uwce37abtm	mortgagestack_calc	replicated 5/5
jpgougoux/calculator		
nmfcyt3u2hcj	mortgagestack_mongo	replicated 1/1
mongo:3.2		
py98oca7s0a1	mortgagestack_notifier	replicated 1/1
jpgougoux/notifier		
t83r6m9q2eex	mortgagestack_optimizer	replicated 1/1
jpgougoux/optimizer		
yqwlzns2fwgd	mortgagestack_reporting	replicated 1/1
jpgougoux/reporting		


Observe que la presencia de los contenedores cuando aumenta la escalabilidad (o su desaparición cuando el número baja), no es inmediato: es Swarm el que controla los contenedores y lo hace según sus propios criterios. Por ejemplo, puede que decida no eliminar inmediatamente un contenedor si se han asignado al cluster muchos recursos. Entre esta manera de funcionar y el tiempo de carga de las imágenes que faltan, el funcionamiento de Docker Swarm pueda parecer algunas veces poco transparente, pero los comandos que permiten ver el estado de diferentes

servicios desmiente esta impresión. Y sobre todo, sea cual sea su manera de actuar, se las arregla para llegar siempre al estado deseado por la aplicación, que es lo fundamental para la robustez de los despliegues.

Mirando directamente a los nodos del cluster, vemos que Swarm ha repartido lo mejor posible los cuatro contenedores adicionales:

images/06RI18N.png

images/06RI19N.png

images/06RI20N.png

De nuevo, no hacemos normalmente este tipo de operaciones en producción, visto que el objetivo es justamente abstraerse de esta complejidad y dejar que Swarm se encargue de escalar a 5, sin tener que hacerse la pregunta de cuál es la mejor manera para añadir una instancia, ni gestionar el hecho de que los contenedores se deban mover para mejorar el reparto, que es necesario detenerlos de manera limpia, etc.

## Alguna información adicional sobre Swarm

Hay mucho que explicar sobre la riqueza de funcionamiento de Swarm. Para dar algunas pistas al lector y permitirle adivinar toda esta riqueza, mencionaremos rápidamente a continuación alguna de estas funcionalidades, sin entrar en detalle.

### 1. ¿Qué sucede en su interior?

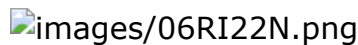
En primer lugar, en el ejemplo que se ha mostrado anteriormente, conviene volver sobre varios puntos que facilitan la comprensión de lo que realmente ha pasado.

#### a. Comandos de diagnóstico

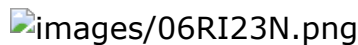
Cuando se lanza el stack, es posible seguir el estado de los servicios, utilizando el comando `docker stack ps`, seguido del nombre del stack destino (la lista se puede encontrar con el comando `docker stack ls`). La siguiente visualización se corresponde, por ejemplo, con el estado al cabo de algunos segundos, con los servicios que han alcanzado su estado `Running` (que se están ejecutando), mientras que otros todavía están en el estado `Preparing` (en preparación, es decir en descarga o en ejecución):

images/06RI21N.png

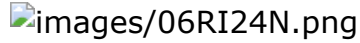
Cuando todo está listo, la visualización pasa a algo parecido a lo siguiente:

images/06RI22N.png

Y cuando el escalado ha terminado, naturalmente encontramos tareas adicionales. La columna `NODE` nos permite ver en qué nodo se han asignado, sin tener que conectarse a la máquina como hemos hecho para facilitar la comprensión:

images/06RI23N.png

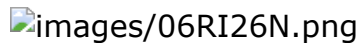
Más allá del stack, también es posible observar precisamente los servicios con el comando `docker service ls`, que devolverá globalmente lo mismo en caso de un despliegue de un stack único, pero con información diferente:

images/06RI24N.png

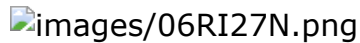
Una vez que se ha localizado un servicio a través de su identificador o nombre, hay otros comandos accesibles como `docker service ps`, que permite recuperar la información de instanciación del servicio:

images/06RI25N.png

El comando `docker service inspect` va a entrar mucho más en detalle, tradicionalmente se usará con un `grep` o como mínimo con un `head` o un `tail` para filtrar la visualización.

images/06RI26N.png

Desde las versiones más recientes de Docker, también es posible consultar los logs remotos, utilizando el comando `docker service logs`, como se muestra a continuación:

images/06RI27N.png

En resumen, como se prometió, es posible hacer casi todo lo necesario para mantener la aplicación en condiciones operativas desde un cliente Docker exterior. Solo los administradores de sistema tendrán necesidad tocar las máquinas que componen el cluster y conectarse a ellas, incluso únicamente en remoto para unirse al cluster.

## **b. Gestión de la red overlay y de los puertos**

De nuevo, para simplificar no vamos a hablar de la capa de red que Docker implementa para facilitar la utilización de Swarm sino que, además de la red dedicada que permite a las máquinas del cluster hablar entre ellas de manera aislada, se crea por defecto una red de tipo overlay por defecto para el stack que se ha desplegado. Se puede ver por el siguiente comando:


images/06RI28N.png

Docker ha creado automáticamente la red llamada `mortgagestack_default`, durante la ejecución del stack, de manera que los servicios se puedan comunicar entre ellos de forma completamente estanca respecto a otras instancias aplicativas. De este modo, por defecto, todos los contenedores de la aplicación lanzada se verán los unos a los otros, pero no verán el resto de contenedores del mismo cluster Swarm, y no serán vistos por ellos tampoco. Esta manera de proceder es muy segura y, si además no es suficiente para sus necesidades aplicativas y dentro del stack desea crear grupos de contenedores que solo se pueden comunicar por un canal dado (por ejemplo, un conjunto front office y un back-office), puede hacerlo creando dos redes separadas en el archivo Docker Compose y asignando cada uno de los servicios a una o varias redes. Si el nivel de seguridad lo requiere, también es posible ir todavía más allá, estableciendo una red cifrada con este tipo de comando:

```
docker network create --opt encrypted --driver overlay redsaneada
```

Un lector curioso puede que se haga la pregunta de la exposición del puerto 8080 al exterior. Después de todo, hemos especificado que el puerto 8080 se debería exponer por el servicio `app`, pero nadie sabe con antelación sobre qué máquina se va a arrancar el contenedor correspondiente. Y sobre todo, solo el primer nodo del cluster se ha expuesto en una dirección DNS externa. Entonces, ¿cómo se hace para que la aplicación funcione correctamente en todos los casos? La respuesta se debe buscar en el lado de la orquestación de la red de Docker Swarm. Durante la implementación de una aplicación con servicios que exponen puertos, Docker implementa lo que se llama un Routing Mesh, es decir, una malla de enrutamiento de mensajes que hace que todo servicio que llame a otro servicio sobre un puerto dado, se verá automáticamente enviado sobre la implementación correspondiente en la máquina correcta. Todavía mejor, Docker gestiona automáticamente el load-balancing si se han lanzado varias instancias del servicio sobre varios nodos del cluster, y se han inscrito entradas DNS con los nombres de los servicios para que su utilización sea más sencilla para el resto. Para terminar, la red llamada `ingress` (ver captura de pantalla más atrás), se encarga de la exposición del puerto fuera del cluster. Al final, Docker ha realizado un montón de operaciones de conexión y orquestación relacionadas con la red, para conseguir que los desarrolladores se puedan concentrar en una única cosa: escribir código sencillo, donde las dependencias entre servicios se hacen simplemente llamando al nombre del servicio y su puerto.




 Este concepto de Routing Mesh se está extendiendo, con productos adicionales como vulvcand.

## 2. Para ir más allá

### a. Detener la plataforma

Después de este ejercicio de despliegue de la aplicación de micro-servicios ejemplo en un cluster Swarm, es visible que Docker ha alcanzado una excelente separación de las responsabilidades. En efecto, para pasar de un archivo Docker Compose que permite describir nuestra aplicación y desarrollarla sobre un demonio equivalente para el despliegue sobre el cluster, lo único que hemos tenido que cambiar al final, es el número de versión, para aumentarlo de 2 a 3. Todo sigue siendo compatible con las antiguas funcionalidades. Respeto a los enlaces, se ha sustituido ventajosamente por una orquestación de red no únicamente segura y con buen rendimiento, sino además completamente automatizada por Docker. En resumen, los esfuerzos son mínimos para desarrollar una aplicación de micro-servicios de manera extremadamente elástica.

Habiendo presentado esto, ahora podemos proceder a parar la instalación, comenzando por el stack lanzado, con ayuda del comando `docker stack rm`:

 `images/06RI29N.png`

Respecto al cluster Swarm en sí mismo, el comando `docker swarm leave` permite sacar un nodo del cluster (es decir, el inverso del comando `join`). De inmediato, las eventuales cargas de trabajo se reproducirán en los nodos restantes. De esta manera, para el caso particular del último nodo de tipo manager, se debe utilizar una opción de forzado `-f`, porque quitarla implica la desaparición del cluster en su conjunto y por tanto, de todo lo que tenía a nivel de aplicaciones. No entramos en los detalles sobre el resto de formas para hacer evolucionar un cluster Swarm.

### b. Conexión a un registro privado

Una observación sobre el ejercicio que se acaba de realizar, es que el registro utilizado era público, pero este enfoque no es

siempre posible con las aplicaciones industriales. Hemos comprobado que el administrador de la aplicación no ha lanzado en ningún momento el comando `docker pull`, ya que es Swarm el que ha distribuido las tareas entre los diferentes nodos. La descarga de las imágenes ha tenido lugar necesariamente, pero de manera transparente. Si el registro utilizado hubiera sido privado, ¿cómo hubiera sido posible esta operación, teniendo en cuenta que el nodo no sabe la contraseña o, de manera general, las credenciales de acceso al registro?

Docker ha pensado en este problema desde las primeras versiones de la implantación de Swarm. Ofrece una opción `--with-registry-auth` para el comando `docker swarm deploy`, que va a utilizar la autenticación local al registro (necesario en el caso de un registro privado para operar el comando `docker-compose push`, como se ha realizado más atrás) y propagarla en los diferentes nodos.

### **c. Gestión de las restricciones**

Otro mecanismo interesante que hay que mencionar aunque sea brevemente, es el de las restricciones. Las restricciones describen las asociaciones entre las etiquetas que tienen los demonios Docker y las etiquetas que tienen los servicios. De esta manera, por ejemplo es posible establecer una restricción para que los contenedores correspondientes a un servicio en particular se arranquen en una máquina concreta del cluster Swarm.

Esto puede servir si las máquinas están equipadas con discos duros rápidos y la aplicación utiliza bases de datos, porque estas últimas serán más eficaces sobre las primeras, mientras que el resto de servicios se beneficiarán de menos ventajas. En el caso de un cluster híbrido, es decir mezclando demonios Windows y demonios Linux (una posibilidad reciente de Swarm), la orquestación de las restricciones también es obligatoria, porque las imágenes de una arquitectura no se ejecutarán en otra (la excepción eran las imágenes multiplataformas, que son una enésima funcionalidad avanzada de Docker, que solo vamos a mencionar para no desviarnos demasiado del tema central del libro).

Por lo tanto, la orquestación de las restricciones es un tema importante para el administrador de aplicaciones Docker, y se recomienda pensar bien asignar las etiquetas correctas a las imágenes Docker, incluso si en un primer momento no se habían

previsto para funcionar en un cluster. Estos metadatos se pueden revelar como imprescindibles.

#### **d. Despliegue incremental (rolling update)**

Una última funcionalidad que el autor desea mencionar sobre Docker Swarm, es el rolling update. La actualización incremental de servicios es una capacidad realmente impresionante de Docker. Consiste en actualizar la versión de los servicios, garantizando que la funcionalidad no se detendrá jamás completamente.

De esta manera, si un servicio debe pasar de una versión a otra de su imagen, el comando a utilizar es `docker service update`. Por defecto, este comando simplemente va a detener todas las tareas correspondientes al servicio y las va a volver a lanzar con la nueva definición de la imagen para los contenedores. Habrá un descenso del servicio, porque todo se detiene al mismo tiempo.

Si hay varias instancias del servicio ejecutándose, es posible especificar un grado de paralelismo de la actualización con la opción `--update-parallelism`, el valor entero especifica el número de tareas a actualizar al mismo tiempo. De esta manera, en el servicio `calc` de nuestro ejemplo, la utilización del valor 2 para esta opción, nos garantizaría que, en las 5 réplicas de este servicio, 3 siempre permanecerán activas durante la actualización incremental. Para esto, es necesario especificar también un plazo entre cada actualización. Esto lo permite la opción `--update-delay`. En nuestro ejemplo, un valor de 30 s (medio minuto) sería ampliamente suficiente. Una instancia de servicio basada en la imagen `calculator` se detendría muy rápidamente, descargaría la actualización y arrancaría (excepto en algún caso particular, algunos segundos). Esta actualización, controlada por Swarm, también permite el mecanismo de rollback, es decir, volver a la versión funcional anterior en caso de problemas. Para resumir, hay un campo de funcionalidades avanzadas de Docker. Es posible percibir su extensión ejecutando un sencillo comando que lista las opciones disponibles para el comando `docker service update`:

```
jpg@VM-UBUNTULTS:~/Git/mortgage$ docker service update --help
Usage:          docker service update [OPTIONS] SERVICE
Update a service
```

## 7. IR AÚN MÁS ALLÁ CON DOCKER

### Docker en su fábrica de software

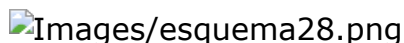
#### 1. Docker a todos los niveles

Hemos mostrado en el capítulo anterior cómo utilizar Docker para desarrollar una aplicación. Pero Docker también es muy útil en los editores de software: participa del funcionamiento fluido de la ALM (*Application Lifecycle Management*, es decir, la orquestación del ciclo de vida del software, desde el diseño a la obsolescencia), integrando una gran cantidad de formas en estas herramientas.

Los puntos de integración son múltiples. Docker es una especie de navaja suiza que va ayudar a resolver diversos puntos de dificultad:

- Producir imágenes en salida de build;
- soportar máquinas de compilación;
- utilizar contenedores para las pruebas;
- establecer más fácilmente una integración continua;
- etc.

En esta segunda parte del capítulo, vamos a centrarnos en estos usos quizás más destinados a los administradores y administradores de herramientas para desarrolladores. El siguiente esquema muestra las diferentes etapas típicas de un ciclo de vida de una aplicación de software, desde el diseño del código fuente a su despliegue, marcando las etapas del uso de Docker que hemos mostrado hasta ahora:



Estos dos usos se corresponden con:

- La implantación de Docker para facilitar el despliegue en un entorno destino, controlando la equivalencia completa entre los entornos, ya sea de desarrollo, pruebas o producción. Esto se corresponde con la flecha entre la herramienta de instalación y los entornos, así como estos últimos, que utilizan Docker.

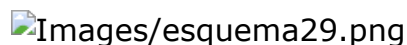
- La utilización de Docker para implementar un registro de imágenes Docker. Para ser preciso, hemos explicado el funcionamiento de un registro como almacén de imágenes Docker, sobre todo en el caso de un registro público. En lo que respecta al registro interno, el funcionamiento es el mismo, salvo que el registro no se expone al exterior (al contrario que un registro público o privado, este último expuesto al exterior pero securizado, mientras que el registro público está abierto a cualquier potencial usuario).

En las secciones siguientes, vamos a ver que Docker puede servir en casi todas las otras etapas de la fabricación de software.

## 2. Producir imágenes como salida de build

### a. Posicionamiento

Una de las primeras utilidades de Docker en la fabricación de software consiste en producir imágenes Docker en la salida, en lugar de entregables "estándares", que después servirán para desplegarse con una herramienta de instalación. Esta tarea se sitúa a nivel de la ALM como podemos ver en el siguiente esquema:



### b. Herramienta

El interés es que la automatización de la tarea fuerce a preparar los `Dockerfile` al mismo tiempo que el código fuente, lo que permite no olvidar los puntos importantes como la apertura de puertos, la declaración de un volumen, etc. Estos puntos son conocidos por el desarrollador del servicio, pero si hay un operativo encargado de la generación del `Dockerfile`, hay más riesgo de olvidarlo fácilmente.

La automatización también es interesante para ganar tiempo y la reducción del número de errores en la compilación de las imágenes. Permite que el entregable salga de la fábrica de software en forma de "claves listas para usar".

Por el contrario, este aspecto práctico va en detrimento de la seguridad: algunas veces, el usuario de sus contenedores desearía saber cómo están creados y estar seguro de qué hay dentro. Evidentemente, cuando es usted el que establece el

entorno en sus propios contenedores, no hay lugar a la pregunta sobre la confianza: en este caso, solo queda gestionar que la seguridad del modo en que se transportan las imágenes de su fábrica de software al entorno destino. Pero cuando el destino no es el suyo, el destinatario está en posición de exigir un nivel de garantía sobre la exactitud de la imagen. Para esto, en las últimas versiones, el registro Docker ha empezado a establecer un sistema de hash para validar que la imagen recibida es exactamente la esperada.

### **c. Consejos de implementación**

La primera etapa, la más sencilla e inmediata, consiste en añadir un `Dockerfile` en los almacenes de código. El posicionamiento en la raíz es lo más lógico.


A continuación, es posible gestionar la llamada de `docker build` a mano, utilizando un script dedicado en su ALM. Hay plug-ins probados para realizar esta operación, por ejemplo en Jenkins. El artículo disponible en el enlace <https://www.wouterdanes.net/2014/04/11/continuous-integration-using-docker-maven-and-jenkins.html>, es un excelente punto de partida.

Una alternativa, si su código puede ser público, es utilizar el sistema de webhooks (técnica que permite asociar llamadas de URL a eventos), para generar automáticamente imágenes Docker, durante las modificaciones de su código en una cuenta GitHub o Bitbucket, como se ha mostrado en el capítulo sobre la creación de sus propias imágenes. La externalización total de la fábrica de software es una fuente de ahorro importante si su modelo de propiedad intelectual lo permite.

## **3. Utilizar contenedores para la fábrica de software**

### **a. Posicionamiento**

Docker también es muy útil para alojar todos los procesos de una fábrica de software. En este caso, su uso se corresponde con las siguientes zonas:

 Images/esquema30.png

## **b. Herramienta**

Una de las primeras cosas interesantes de desarrollar todos los servidores de orquestación de código fuente, de integración continua, de build, etc., es la flexibilidad de la ALM. De esta manera es posible aumentar muy fácilmente el número de máquinas de build, y esto utilizando mejor los recursos a través de la implantación de máquinas virtuales (más pesadas) - incluso físicas (más pesadas y complejas de instalar). La escalabilidad se simplifica mucho.

Pero también es posible sellar porciones de una ALM, allí donde el sobrecoste en infraestructura lo hace imposible. La separación clara de porciones de ALM es un buen método para permitir una migración sin problemas a un enfoque de propiedad intelectual diferente.

Para terminar, el reinicio después de un incidente sobre una arquitectura de contenedores, también se mejora por la capacidad de los contenedores de establecerse muy rápido en un cluster, incluso remoto (la ALM utiliza muchos recursos y ancho de banda y su posicionamiento en local o en un cloud privado, todavía es la opción mayoritaria, porque favorece los mejores rendimientos).


Recientemente, han aparecido numerosas herramientas para la orquestación de la ALM para Windows, así como herramientas de superposición del despliegue o de orquestación de los contenedores como Rancher. Es posible establecer una ALM completa en el cloud. Esto favorece mucho la compartición de las buenas prácticas. De manera demasiado habitual, la implementación de una ALM todavía era una actividad artesanal y se beneficiaba mucho de la industrialización aportada por una estandarización de los módulos y de su interacción por su funcionamiento en los contenedores.

## **c. Consejos de implementación**

La realización de los `Dockerfile` no plantea ninguna dificultad particular, las aplicaciones que componen una ALM son, en general, aplicaciones de servidor como el resto, que exponen APIs, que utilizan recursos de archivos y en particular archivos temporales. Estos últimos se ubicarán ventajosamente en un volumen dedicado, que posteriormente se puede llevar fácilmente a discos duros rápidos para mejorar el rendimiento del conjunto. Sobre todo, estos espacios de almacenamiento pueden ser menos

robustos (la pérdida de archivos temporales no es grave) y con un coste menor.

El único consejo que se puede dar en esta sección es el mismo que en la mayor parte de los usos de Docker, a saber, comenzar por comprobar si una imagen no existe ya y, si no se trata de una imagen de confianza, validar que el contenido del `Dockerfile` está conforme a lo que se espera.

 Atención, si compila las imágenes Docker en el marco de su build, pero este proceso en sí mismo funciona en un contenedor Docker, está en una situación de tipo "Docker in Docker", que necesita funcionar en modo privilegiado. <https://github.com/jpetazzo/dind>, explica la operación asociada.

## 4. Utilizar los contenedores para las pruebas

### a. Posicionamiento

La fábrica de software tradicional lanza varios niveles de pruebas en salida de build. Las pruebas unitarias y smoke tests (pruebas superficiales pero muy rápidas, realizadas inicialmente antes de baterías de pruebas más profundas, para evitar una pérdida de tiempo durante un defecto importante que afecte a numerosas pruebas), se integran en la build en sí misma, pero las pruebas de integración y las pruebas de sistema se realizan de manera asíncrona, bajo petición o fuera de horario pero, en cualquier caso, antes de una release.

 Images/esquema31.png

### b. Herramienta

Docker puede representar una enorme ventaja para las pruebas. Puede volver a situar un entorno en un estado dado de manera muy sencilla, para lanzar un contenedor en una capa de imagen en modo solo lectura. También es posible implementar fácilmente un entorno de pruebas, lanzar las pruebas que lo van a modificar algunas veces en profundidad y después, simplemente abandonar la capa en modo escritura que se ha escrito y relanzar un contenedor para reproducir exactamente el mismo entorno de pruebas inicial.



En lugar de recurrir a trucos enrevesados como restaurar los archivos básicos de datos o modificar los drivers para que no aplique realmente las modificaciones en las bases de datos, es más sencillo lanzar un contenedor, eliminarlo al final de las pruebas y volver a comenzar desde el inicio.

La velocidad de ejecución de un contenedor es tal, que permite otros usos hasta ahora inimaginables, como restaurar un entorno virgen, no ya entre cada batería de pruebas, sino antes de cada prueba. Un enfoque como este, era imposible en máquinas virtuales, incluso con una orquestación optimizada de los snapshots de VM. Con los contenedores se convierte en algo sencillo.

### **c. Consejos de implementación**

Con el objetivo de formar una base sólida de entornos de pruebas, en primer lugar es conveniente tener un buen dominio de las pruebas en sí mismas. Es inútil querer sacar provecho de los contenedores Docker para mejorar las pruebas, si estas no se han implantado correctamente a partir de casos de prueba detallados, origen de condiciones formales.

Durante la implementación de los entornos, las modificaciones manuales con `commit` de la imagen, son muy prácticas para crear un nuevo entorno para las diferentes pruebas. Por el contrario, se hace rápidamente útil establecer una buena clasificación de estos entornos.

### **d. Variante asociada**

La continuación lógica de la independencia de las baterías de pruebas, consiste en utilizar Docker para montar un entorno de pruebas de integración, incluso para las pruebas de sistema.

 Images/esquema32.png

Aunque es innovador el enfoque de reducir los entornos únicamente al de producción, las nociones de entornos de pre-producción/pruebas/verificación/integración/etc. son más actuales. La capacidad que ofrece Docker de duplicar rápidamente un entorno completo, exportando los contenedores en forma de imágenes, es un buen añadido.

En particular, la estanqueidad de los recursos respecto al host, permite simplificar las operaciones de varios entornos. Para

tomar solo un ejemplo, si un entorno utiliza un puerto, en general es complejo incorporar un segundo entorno copia del primero sin configuración, con el objetivo de no correr riesgos de conflicto de puerto. Esto puede provocar problemas cuando la propagación del cambio de puerto en los servicios llamadores no se domina y un servicio de producción termina por llamarse por otro desde un entorno de pruebas.

La estanqueidad de los contenedores los unos respecto a los otros, así como la capacidad de redirigir de manera muy sencilla un puerto a otro, un directorio sobre otro o incluso cambiar en caliente las variables de entorno, son enormes ventajas en estas situaciones.

## 5. Vuelta al registro

La implantación de un registro de imágenes, ha ocupado una parte importante de este libro, porque se trata de una etapa importante en la adopción de los contenedores por una empresa. Aquí no volveremos sobre la implantación de un registro, sino sobre la integración en la fábrica de software de este registro, para ser más precisos, de estos registros, porque conviene diferenciar entre un registro interno que contiene todas las versiones actuales de las imágenes y un registro público, destinado a explotarse por clientes de la empresa, y que solo exponen las versiones correctamente probadas y validadas.


Para el registro público, hay dos opciones posibles. La primera manera de actuar, la más sencilla, consiste en abrir una cuenta en Docker Hub para publicar sus imágenes. Sencillez y bajo coste son las ventajas, pero será necesaria una cuenta de pago para conservar como privado más de un almacén. Para necesidades sencillas (PME, independientes, etc.), esto puede ser suficiente. Para los usos más industriales, las cuestiones de propiedad intelectual, disponibilidad y reducción de los permisos de acceso, harán que la solución quizá sea crear y gestionar su propio registro.

Del lado del consumidor de las imágenes, también puede ser interesante mostrar un registro privado, que sirva de caché local, de manera que los administradores puedan obtener las diferentes imágenes validadas por sus medios, vengan de un editor de software o del mundo open source. Esta caché sirve potencialmente de memoria RAM, exponiendo solo al consumo interno las imágenes que han sido validadas por el servicio de calidad.

El interés también reside en el hecho de que se reduce el ancho de banda y que la validación del hash se puede realizar de una vez para todas. La corrupción de la imagen por un atacante es un escenario menos probable dentro de la red interna.

Para terminar, las cuestiones de seguridad hacen que en general los servidores de producción no puedan acceder a Internet, lo que hace obligatoria la presencia de un registro local.

La representación de esta opción es la siguiente:

 Images/esquema33.png

## Antes de empezar en producción con Docker

En la actualidad, el ecosistema de Docker es de tal riqueza que un libro sobre los fundamentos del funcionamiento solo puede mencionarlo.

En particular, se podría dedicar un tomo completo a la orquestación de Docker en producción. Sin extenderse sobre un posible contenido, parece imposible no mencionar en un libro sobre Docker, aunque sea brevemente, algunos aspectos como los siguientes.

### 1. Seguridad

Durante mucho tiempo, Docker ha estado rezagado respecto a la seguridad. La empresa ha querido ocupar el mercado lo más rápidamente posible, para no ceder espacio a la competencia y algunas veces, el precio a pagar ha sido la seguridad de las instalaciones. Afortunadamente, esta época ha pasado desde que el software en sí mismo se ha hecho estable y se ha dedicado tiempo a su securización.

En las últimas tecnologías aparecidas en 2016 y 2017, la seguridad se ha pensado desde el mismo momento de la aparición de las nuevas funcionalidades. Este es el caso en particular de la implementación de un cluster Docker Swarm, donde el acoplamiento de un demonio al cluster, solo se hace si el primero conoce un token secreto. Además, el token se divide entre una versión para unirse como sencillo agente de ejecución o para unirse al cluster como nodo de tipo manager.

Hay buenas prácticas que han aparecido sobre la securización de las imágenes, con el uso del comando `USER` para los `Dockerfile` o la opción `--user` en un `docker run`, de manera que no pueda funcionar en `root`. SELinux y App Armor se soportan completamente. En resumen, ahora este aspecto se trata correctamente y no dedicaremos una sección (necesariamente muy larga), para explicar todos los mecanismos de seguridad de Docker. En la página <https://docs.docker.com/engine/security/>, se ofrece a los más expertos todas las pistas necesarias.

### 2. Restricción sobre los recursos

Otro punto principal antes de arrancar en producción con una arquitectura de contenedores, es tomar en cuenta el equilibrado de los recursos, que necesita establecer una cuota. En las arquitecturas de micro-servicios, los contenedores garantizan una gran estanqueidad entre servicios (incluso entre seguidores), y esta estanqueidad solo se puede completar si afecta también a los recursos.

Docker permite limitar el uso de memoria de un contenedor, gracias a la opción `--memory`, así como al uso del procesador, con ayuda de la opción `--cpushares`. Sin entrar en detalles, un comando como este permite limitar los dos recursos:

```
docker run -it --memory 256m --cpushares 256 ubuntu
```

La memoria está limitada en tamaño absoluto, mientras que el límite de las franjas de procesador se expresa de manera relativa. Como el contenedor ejemplo anterior es el único en la máquina host, puede utilizar el 100% de la CPU (limitado a lo que le proporcione el nodo Linux, por supuesto). Pero si un segundo contenedor arranca con un valor de `--cpushares` de 512, por ejemplo, el primer contenedor entonces deberá conformarse con un tercio de las franjas disponibles, mientras que el segundo obtendrá dos tercios. A la inversa, si el valor elegido para el segundo contenedor es de 128, el escenario será el inverso. En efecto, el primero habría recibido 256 partes y el segundo el doble. La compartición se haría según los ratios  $1/3$  y  $2/3$ , siguiendo una regla de proporcionalidad.

## **Docker y Windows**

Aunque Docker sea un producto de Linux y basado en funcionalidades de bajo nivel, que no se encuentran en la misma forma en el nodo Windows, Microsoft ha conseguido en 2016 crear una implementación nativa de Docker, y ha tenido la buena idea de seguir las recomendaciones del Open Container Initiative, que garantían una compatibilidad con Docker a nivel del API. Por supuesto, las imágenes no se pueden mezclar entre las máquinas Windows y Linux, pero el API es el mismo en un caso o en el otro. Esto no solo permite un fácil aprendizaje, sino abrir incluso la vía a escenarios extremadamente potentes, como plataformas Swarm híbridas, compuestas por el demonio de los dos tipos y por lo tanto soportando contenedores Windows y Linux, en la misma plataforma de despliegue.

## Conclusión

Docker ha cambiado la manera en la que lo desplegamos y de rebote, la relación entre los desarrolladores y los administradores de sistema, que ya había evolucionado mucho con el movimiento DevOps.

Hay muchas posibilidades de que haya más sorpresas y ya podemos imaginar algunos cambios adicionales, por ejemplo, la diseminación cada vez más grande de servicios incorporados por los contenedores Docker. La capacidad de Docker de presentar una interfaz idéntica independientemente del soporte de máquina del contenedor, permitirá a determinadas imágenes convertirse en estándares de facto para un servicio en particular. Adicionalmente, esto permitirá a los desarrolladores concentrarse en el valor añadido de los desarrollos, y no tener que reinventar constantemente servicios, como sigue siendo el caso demasiado habitualmente, en la actualidad.

Del lado de la orquestación, Docker ha anunciado en octubre del 2017 el soporte de Kubernetes de Google, que podría parecer algo negativo para Swarm, pero que bien mirado es una evolución lógica del ecosistema. Docker siempre ha tenido un enfoque llamado "pilas incluidas", es decir, que una funcionalidad se podría cambiar, pero Docker proporciona al menos una versión sencilla que permite al "juguete" funcionar desde su salida de la caja.

Swarm es, de una determinada manera, la pila entregada que permite jugar inmediatamente y que, en la mayoría de los casos, es ampliamente suficiente. En efecto, Kubernetes es más sofisticado y podrá realizar hazañas en términos de despliegue pero al final, muy pocas empresas necesitan una potencia equivalente a la de Google y es una apuesta segura que Swarm continuará siendo una elección sencilla para muchos usuarios. El soporte de los dos es un aspecto muy interesante. De la misma manera que el hecho de disponer de un buen coche no quiere decir que ya no necesitemos otro coche.

En relación con otro aspecto más futurista, las versiones de Docker adaptadas a los teléfonos móviles, a los objetos conectados y a otros periféricos, podría añadir una capa de estanqueidad necesaria para estos sistemas donde la seguridad es baja.

El ecosistema Docker burbujea literalmente y esta expansión es, a la vez una fuerza y una debilidad. La riqueza se acompaña de una

incertidumbre sobre las herramientas y las buenas prácticas que se impondrán. De hecho, el lector verá rápidamente que algunas tecnologías mencionadas en este libro quedan obsoletas. De aquí el interés de dedicarse a los conceptos. A la inversa, es una parte completa de la industria de la informática la que se desarrolla ante nuestros ojos y los cambios serán profundos, ciertamente del mismo nivel que los generados por la virtualización hace veinte años o casi.

Afortunadamente, en esta expansión de nuevas herramientas, hay determinadas verdades que permanecen. Por ejemplo, el hecho de que el rendimiento energético depende más del desarrollador que de la herramienta. Docker tiene el potencial para convertirse en una magnífica herramienta de reducción del consumo de recursos en los datacenters. Pero, de la misma manera que la virtualización no ha impedido una explosión de la producción de servidores (normalmente a causa de una infrautilización remanente), hay riesgos reales de que la facilitación del uso de Docker al final conduzca a un consumo de recursos más elevado. Nos corresponde a nosotros asegurarnos de que destaquen los mejores aspectos de la herramienta.