TSR 2023/24

TEMA 4 – DESPLIEGUE DE SERVICIOS. DOCKER

Guía del Alumno

Objetivos

- Describir los aspectos a considerar durante el despliegue de aplicaciones distribuidas.
- Mostrar los problemas derivados de las dependencias y cómo resolverlos.
- Discutir ejemplos concretos de despliegue.

CONTENIDO

1 INTRODUCCIÓN

El objetivo de toda aplicación distribuida es proporcionar servicio a sus usuarios. Sin embargo, ese objetivo no puede alcanzarse tan pronto como se hayan desarrollado los componentes de la aplicación pues estos necesitan ser instalados en varios ordenadores y ser iniciados antes de que sus clientes puedan acceder a ellos.

Citando a Carzaniga et al. [1]...

> "*Software applications are no longer stand-alone systems. They are increasingly the result of integrating heterogeneous collections of components, both executable and data, possibly dispersed over a computer network. Different components can be provided by different producers and they can be part of different systems at the same time. Moreover, components can change rapidly and independently, making it difficult to manage the whole system in a consistent way. Under these circumstances, a crucial step of the software life cycle is* deployment*—that is,* ***the activities related to the release, installation, activation, deactivation, update, and removal of components****, as well as whole systems."*

Por tanto, el despliegue no consiste únicamente en la instalación y activación de los programas sino también en su edición (o publicación), actualización, desactivación y desinstalación. De hecho, el despliegue contiene todas las tareas de la gestión del ciclo de vida de una aplicación informática que suceden tras su desarrollo.

Así, con las fases de análisis, diseño y desarrollo se habrán obtenido algunos componentes de la aplicación, pero esas piezas deben desplegarse para llegar a ser un servicio. Esquemáticamente:

Programas + Despliegue --> Servicios

Revisemos qué ocurre cuando queremos desplegar una aplicación normal de escritorio:

1. Esa aplicación ha sido editada por alguna empresa u organización. Puede ser descargada una vez se haya comprado o puede haber sido editada en algún soporte extraíble (p.ej., en un DVD-ROM).
2. Ahora, debemos instalarla. A primera vista, solo necesitamos copiar sus ficheros ejecutables en alguna carpeta (o conjunto de carpetas) en nuestro ordenador. Sin embargo, este paso es más complejo de lo que parece. Además de copiar los ficheros ejecutables y otra información de configuración, varias dependencias deben ser resueltas (p.ej., algunas bibliotecas estándar, o quizá dependientes del lenguaje de programación utilizado, pueden ser necesarias o algunas herramientas deben encontrarse en el sistema para ejecutar o instalar la aplicación). Si alguna de esas dependencias no pudiera resolverse, las piezas correspondientes deberían buscarse e instalarse correctamente para reanudar el despliegue.
3. Finalmente, cuando el paso anterior haya sido completado satisfactoriamente, esa nueva aplicación podrá ser iniciada y ejecutada.
4. Dependiendo del tipo de aplicación, en su primer inicio (y en comprobaciones periódicas posteriores) estos programas se activarán comprobando frente a un servidor de licencias remoto que posee un identificador o ticket de activación válido.

Ya que el segundo paso puede llegar a ser complejo, varios sistemas operativos facilitan una API y un conjunto de herramientas para desarrollar instaladores de aplicaciones. Por ejemplo, en Windows existe una API [2] para Windows Installer [3] que debe conocerse para construir los paquetes MSI (que son los utilizados para instalar aplicaciones en los sistemas Windows). Entre las herramientas que acompañan a esa API encontramos WiX (Windows Installer XML Toolset) [4] que será la herramienta a utilizar para construir los paquetes MSI. El paquete MSI almacena los componentes de la aplicación y conoce qué otras dependencias deben ser resueltas durante la instalación, reclamando su descarga e instalación en caso de que no estuvieran presentes en el equipo donde se realice el despliegue. Con este soporte, la instalación y actualización de aplicaciones puede automatizarse.

Existen herramientas y API similares en otros sistemas operativos. Por ejemplo, las distribuciones Linux Fedora utilizan paquetes RPM (RPM Package Manager) [5] y las herramientas DNF [6] para este fin.

2 SERVICIOS

En la sección anterior hemos visto que para facilitar servicios será necesario desplegar aplicaciones. En el caso de los sistemas distribuidos, esas aplicaciones están formadas por múltiples componentes, es decir, por programas que implantan una funcionalidad determinada. Esos componentes deben ser autónomos. Eso significa que cada uno de ellos facilitará, a su vez, servicios a los demás componentes. Con un diseño adecuado, los componentes serán reutilizables y proporcionarán funcionalidades de uso general que podrán ser aprovechadas en otras aplicaciones futuras.

Por tanto, si se consideran las tareas de despliegue, un componente...

- Es la unidad de despliegue de aplicaciones distribuidas. Cada componente es un programa que puede depender de otros componentes para construir servicios de mayor capacidad. A la hora del despliegue, el código de cada componente se mantiene normalmente en un paquete (o BLOB) independiente.
- Se puede instanciar tantas veces como se necesite (generando una réplica del componente en cada una de sus instancias) para soportar la carga de trabajo prevista.

Cuando se desplieguen múltiples instancias de cada componente, entonces cada instancia...

- Puede ser iniciada o parada independientemente de las demás instancias. Por regla general, se iniciarán nuevas instancias cuando la carga del componente se incremente y se pararán instancias existentes cuando no haya suficiente carga a repartir entre ellas. Así los componentes pueden adaptarse dinámicamente al nivel actual de carga de trabajo, minimizando el uso de recursos y el consumo energético cuando se reduzca la carga y usando una cantidad ajustada de recursos cuando la carga crezca.
- Puede fallar independientemente de las demás, puesto que cada instancia debería instalarse en un ordenador diferente. Las instancias suelen fallar cuando hay algún problema en su entorno, como pueda ser un fallo en el ordenador que las mantenga.

La **Figura 1** muestra un ejemplo de despliegue de un servicio distribuido. En él, el servicio consta de tres componentes (C1, C2 y C3). Cada uno tiene un número diferente de instancias. C1 mantiene tres instancias, C2 tiene dos y C3 solo tiene una.

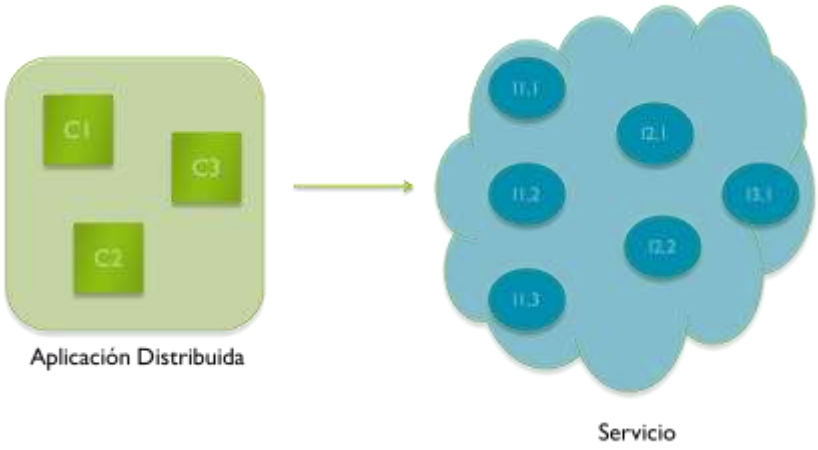


Figura 1. Despliegue de un servicio distribuido

Si el diseño de la aplicación ha sido cuidadoso, esos tres componentes estarán débilmente acoplados. Eso significa que cuando una operación del componente C_i sea invocada no se necesitará invocar a otros componentes para servirla o, en caso de que se necesitara alguno, se realizarían muy pocas invocaciones para ello y estas no implicarían el intercambio de mucha información. De esta manera, el administrador tendrá libertad durante el despliegue para elegir en qué ordenador se instalará cada instancia.

Si en lugar de esto se hubieran utilizado componentes fuertemente acoplados tendríamos un mal diseño. Por ejemplo, imaginemos que los componentes C2 y C3 tuvieran una interdependencia fuerte (es decir, cuando se invoque una operación en uno de ellos, su servicio implicará la invocación de varias operaciones del otro componente y en cada invocación se necesitará transferir mucha información). En ese caso, cada instancia de C2 estaría fuertemente acoplada con cada instancia de C3. Por ello, estaríamos condenados a desplegar estos componentes a la par; es decir, si la primera instancia de C2 se instalara en un nodo N1 entonces la primera instancia de C3 también debería instalarse en N1, si la segunda instancia de C2 se instalara en N2, la segunda instancia de C3 también estaría en N2, y así sucesivamente. Por ello, C3 debería tener el mismo número de instancias que C2 y cualquier variación en el número de instancias en alguno de los dos componentes obligaría a realizar el mismo cambio en el otro componente.

2.1 Acuerdos de nivel de servicio (SLA)

Tras desplegar un servicio, este puede ser utilizado por múltiples usuarios. En caso de servicios distribuidos, el proveedor de servicios suele ser una empresa especializada y sus clientes pueden ser también empresas con sus respectivos usuarios finales (que pueden ser tanto internos como externos a esa empresa cliente). En el área de los servicios distribuidos se suelen establecer contratos o acuerdos de nivel de servicio (SLA, por sus siglas en inglés) entre el proveedor de servicios y sus clientes.

En un SLA se consideran múltiples aspectos, relacionados con:

- La **funcionalidad** facilitada. Se debe asegurar que el servicio facilitado por el proveedor se ajusta a las necesidades y requisitos establecidos por sus clientes.
- **Rendimiento**. Como el rendimiento suele depender de la carga, los clientes solicitarán un nivel mínimo de rendimiento, pero el proveedor podrá también solicitar a los clientes que no excedan cierto nivel máximo de carga.
- **Disponibilidad**. El proveedor deberá garantizar que el servicio estará disponible cuando los clientes lo necesiten. Para ello, un nivel mínimo de disponibilidad (expresado como el porcentaje de tiempo en que el servicio estará accesible, respondiendo toda solicitud recibida) deberá establecerse en el SLA.

Para reflejar estos tres aspectos (y cualesquiera otros que se consideren relevantes) se definen algunas métricas y sus rangos de valores aceptables (conocidos como objetivos de nivel de servicio, o SLO, por sus siglas en inglés) deben ser acordados entre proveedores y clientes.

En el campo de los servicios distribuidos, una de las primeras especificaciones de SLA fue generada por IBM para los servicios web [7].

2.2 Tipos de servicio

En el área informática se pueden distinguir dos tipos principales de servicios:

- **Efímeros**: Cuando los servicios solo se necesiten durante sesiones breves, normalmente interactivas. Los servicios efímeros son implantados por aplicaciones de escritorio a desplegar en ordenadores personales, utilizados en sesiones por parte de un solo usuario. Ejemplos: procesadores de texto, navegadores web, intérpretes de órdenes...
- **Persistentes**: Estos servicios necesitan estar continuamente disponibles y serán utilizados concurrentemente por multitud de usuarios mediante conexiones remotas. Es el caso habitual en los servicios distribuidos. Ejemplos: banca electrónica, administración electrónica, comercio electrónico, etc.

3 DESPLIEGUE

En el despliegue inicial (es decir, en la edición, instalación e inicio) de una aplicación distribuida se pueden distinguir los siguientes pasos:

1. Decidir qué servicios dependientes van a ser utilizados cuando la aplicación se esté ejecutando. En este paso necesitamos establecer las dependencias que surgirán cuando los componentes de nuestra aplicación empiecen a funcionar. El SLA para esos otros servicios debe ser considerado en este paso, estableciendo sus SLO y analizando sus costes.
2. Decidir qué componentes van a utilizarse. Durante la etapa de diseño se habrá delimitado un conjunto de componentes. Algunos de ellos pueden proporcionar funcionalidad opcional o implantarán soluciones alternativas para un mismo problema. Durante este paso, se elegirá el conjunto de componentes a desplegar, dependiendo de los requisitos de sus usuarios. Además, el administrador deberá decidir cuántas instancias de estos componentes habrá que instalar inicialmente.

3. Decidir en qué conjunto de nodos se ejecutará cada componente. Esto dependerá de si existen algunos componentes fuertemente acoplados (aspecto a evitar, como ya hemos explicado anteriormente), de la capacidad de cada nodo y de los requisitos de ejecución que imponga cada tipo de componente.
4. Llevar el código de un componente a los nodos en los que las instancias de ese componente deban ser ejecutadas. Hay herramientas (facilitadas por la plataforma, el middleware o el sistema operativo) que automatizan la transmisión de los ficheros hasta los nodos destino, así como su instalación parcial.
5. Decidir el orden en que los componentes serán iniciados. Ya que los componentes pueden tener algunas interdependencias, esas dependencias deben ser consideradas para establecer ese orden de inicio.
6. Configurar cada componente. Las configuraciones dependen, de nuevo, de las dependencias que existan. Por ejemplo, si el componente A necesita utilizar operaciones de los componentes B y C, los puntos de acceso (o "endpoints") de B y C deben ser comunicados a A, fijando valores para un conjunto de parámetros en su fichero de configuración.
7. Utilizar el sistema operativo de cada nodo para iniciar las instancias de los componentes. Esto debe realizarse de acuerdo con el orden de inicio decidido en el paso 5.

Pero el despliegue no solo consiste en esos pasos. Una vez el servicio esté en ejecución, habrá otras tareas relacionadas con el despliegue que tendrán que realizarse. Esas otras tareas son:

- **Gestión de los fallos** en las instancias de los componentes. Mientras se esté ejecutando un componente, algunas de sus instancias podrán fallar. Esos fallos deben permanecer ocultos para el resto de componentes (y para los usuarios, en caso de que el componente proporcione la interfaz de acceso al servicio). Para ello se utilizarán técnicas de replicación y mecanismos de detección de fallos. Esto implica que las réplicas defectuosas serán paradas y reiniciadas una vez el defecto o error que haya generado el fallo haya sido corregido. Adicionalmente, durante el intervalo de recuperación, el resto de réplicas habrán podido modificar su estado y esas modificaciones deben ser propagadas a la réplica que se esté recuperando para que ésta las considere y aplique, utilizando para ello protocolos de recuperación adecuados.
- **Actualización del *software*** (también conocida como "reconfiguración del *software*"). De vez en cuando los componentes necesitan ser actualizados. Algunas razones para ello son: (1) la detección de errores en los programas, necesitando una nueva edición de esos programas que corrija los errores detectados, (2) los requisitos de los usuarios pueden cambiar y los programas deben ser adaptados para satisfacer esos nuevos requisitos, (3) nuevas tecnologías (más eficientes) han podido surgir, con las que sería posible mejorar el rendimiento o minimizar el consumo de recursos de los componentes, etc. Si esto conduce a una actualización de los componentes, sus SLA deben ser reconsiderados durante la actualización. Para ello, como los componentes estarán replicados, debería diseñarse algún procedimiento que permita a ambas versiones del *software* (la antigua, a retirar, y la nueva, a desplegar) coexistir mientras se transfiera el estado desde la vieja a la nueva versión, sin interrumpir el servicio. Existen algunas soluciones de este tipo, descritas en [8].
- **Mantener la escalabilidad**, adaptándola a la carga (es decir, proporcionar *elasticidad* [9]). Cuando se incremente la carga, los componentes del servicio deben incrementar

su capacidad de servicio ("scale out"), iniciando más instancias para ello. Por el contrario, cuando la carga decrezca, algunas de las instancias previamente iniciadas tendrán que parar ("scale in"). En caso contrario, se estarían utilizando más recursos de los realmente necesarios. Esta escalabilidad adaptativa (o elasticidad) necesita un subsistema de monitorización adecuado que vigile los valores actuales de algunas métricas (p.ej., el tiempo de respuesta) y que reaccione cuando los valores de estas métricas excedan ciertos umbrales (p.ej., añadir una nueva instancia si el tiempo de respuesta superase los 50 ms y eliminar alguna de las existentes si el tiempo medio de respuesta fuese inferior a 2 ms).

3.1 Entradas iniciales

En la Sección 1 vimos que para desplegar una aplicación de escritorio se necesitaba construir cierto paquete de *software*. Ese elemento se desplegará utilizando alguna herramienta dependiente del sistema operativo subyacente. Para desplegar una aplicación distribuida se necesitará utilizar más de un ordenador. Por tanto, el escenario resultante será bastante más complejo y otros elementos serán necesarios para desplegar un servicio. Revisemos cuáles serán las entradas a considerar en este proceso:

- **Aplicación distribuida**. Básicamente, una aplicación distribuida consta de un conjunto de programas (uno por componente) y un conjunto de datos de configuración. En una revisión detallada, podríamos distinguir estos elementos:
    - Cada componente tiene su propio BLOB de código; es decir, un fichero (o parte de un fichero) que mantiene el programa a ejecutar por el componente.
    - Una plantilla de configuración por cada componente. Además de su BLOB, cada componente necesita alguna información de configuración. Esa información se utilizará para rellenar una plantilla de configuración, especificando los valores para múltiples parámetros de configuración.
    - Cierta descripción de las dependencias de cada componente debe ser proporcionada.
    - Plantilla para un plan de interconexión de componentes durante el despliegue. Esta plantilla contendrá los elementos que se listan a continuación y especifica la secuencia de acciones de resolución de dependencias a realizar durante el despliegue:
        - Lista de dependencias que deben resolverse durante el despliegue, especificando los puntos de acceso (o "endpoints") todavía libres y sus clientes potenciales.
        - Lista de puntos de acceso globales que se proporcionarán como puntos de accesos públicos del servicio (es decir, aquellos a utilizar por los usuarios del servicio y no por otros componentes internos).
    - Configuración global de la aplicación. Esta configuración puede ser gestionada mediante diferentes políticas. Dependiendo de la política elegida, se asignarán diferentes valores a algunos de los parámetros de las plantillas de configuración de los componentes, así como para rellenar la plantilla del plan de despliegue.

- **Descriptor de despliegue**. Un descriptor de despliegue contiene:
    - Plantillas de configuración rellenas. Habrá tantas plantillas rellenas como componentes se necesiten.
    - Plantilla del plan de despliegue rellena. Debe observarse que esta plantilla no puede rellenarse mientras no se haya decidido dónde ubicar (es decir, en qué nodo) cada instancia de cada componente. Una vez se conozcan las direcciones y puertos en los que ubicar las instancias, se rellenará esta plantilla.
    - Resolución de dependencias internas. Está implícita en el punto anterior, al establecer la ubicación de cada instancia.
    - Resolución de dependencias externas. Algunos de los componentes a utilizar en el despliegue habrán sido desarrollados por otras empresas y puede que ya estén desplegados en el sistema distribuido. En ese caso, los puntos de acceso de esos componentes "externos" deben ser localizados durante el despliegue y anotados en el descriptor tan pronto como sea posible. Un mecanismo para lograr esta resolución dinámica está basado en el uso de un servidor de nombres. En ese caso, el descriptor de despliegue mantiene los nombres de los componentes externos y sus correspondientes puntos de acceso se averiguarán resolviendo dichos nombres.

3.2 Ejemplo de despliegue

La **Figura 2** muestra un ejemplo de despliegue de una aplicación distribuida.

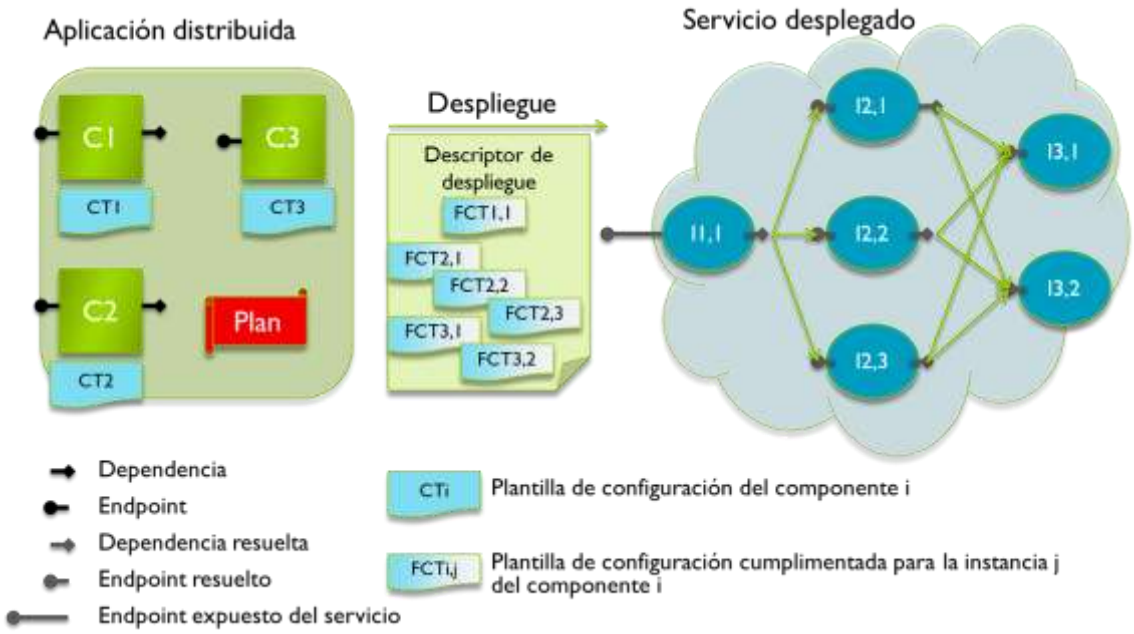


Figura 2. Ejemplo de despliegue.

En este ejemplo, la aplicación a desplegar tiene tres componentes:

- C1 (equilibrador de carga). El equilibrador de carga asigna cada solicitud entrante a una de las instancias del componente C2.
- C2 (lógica de negocio). Procesa cada solicitud recibida. Cuando una solicitud modifique los datos de la aplicación, un protocolo de replicación del almacén de datos será utilizado por C2 para decidir a qué instancias del componente C3 se redirigirá la petición.
- C3 (almacén de datos). Mantiene los datos persistentes del servicio. Cada instancia de C3 no sabe que está replicada. El protocolo de replicación de C3 es gestionado por las instancias de C2.

El plan de despliegue establece que el único punto de acceso público al servicio es el facilitado por C1 y también indica que no hay dependencias externas.

El descriptor de despliegue indica que hay: una única instancia de C1, tres instancias de C2 y dos instancias de C3. También contiene las plantillas de configuración rellenas para cada instancia.

Las dependencias han sido resueltas en el descriptor de despliegue como sigue. La única dependencia de la instancia C1 se corresponde con los puntos de acceso de cada instancia de C2. Cada dependencia de cada instancia de C2 se relaciona con el punto de acceso de cada instancia de C3.

Como los componentes de esta aplicación están débilmente acoplados, cada instancia de cada componente puede ser ubicada en un ordenador diferente. Las direcciones de esos ordenadores se utilizan para rellenar el descriptor de despliegue.

Utilizando el descriptor de despliegue, la aplicación se instala finalmente en los nodos especificados en el descriptor.

4 CONFIGURACIÓN: INYECCIÓN DE DEPENDENCIAS

Uno de los pasos iniciales del despliegue es la generación del descriptor de despliegue. Para ello, las plantillas de configuración de cada componente deben rellenarse tantas veces como instancias vaya a tener ese componente, en función de los recursos existentes en cada uno de los ordenadores que hospedarán a esos componentes.

Parte de la información a utilizar para rellenar esas plantillas está relacionada con la resolución de dependencias. Cada componente debe ejecutar cierto programa y ese programa exporta uno o más puntos de acceso (los necesarios para utilizar las operaciones que exporte) pero también tiene algunos puntos de dependencia (es decir, aquellos fragmentos de su programa desde los que se invocan operaciones facilitadas por otros componentes). Los puntos de dependencia rara vez podrán resolverse durante la compilación del programa. Su resolución se realizará durante el despliegue o durante la ejecución, respetando unos principios similares a los empleados en las bibliotecas de enlace dinámico [10]. En el enlace dinámico las bibliotecas son cargadas en memoria bajo demanda y la dirección a utilizar por el invocador para acceder a una operación de la biblioteca se toma de una tabla de indirección que ha sido rellenada al cargarse la biblioteca por primera vez. Por tanto, la resolución se retrasa tanto como sea posible (resolución dinámica) y sin requerir ningún cambio en la forma en que se escribe el código de las bibliotecas o el código

de los programas que las utilicen (transparencia para el programador). Esos son los principios a respetar a la hora de gestionar el enlace entre las dependencias y los puntos de acceso de los componentes: resolución dinámica y transparencia.

¿Cómo puede obtenerse esa transparencia? Revisemos el ejemplo de las bibliotecas de enlace dinámico para obtener algunas pistas. Si asumimos que una primera versión de la biblioteca a utilizar pudo basarse en enlace estático, el código de aquella primera versión no habrá necesitado ningún cambio para usar enlace dinámico. Entonces... ¿dónde se han aplicado los cambios? Básicamente, en la herramienta utilizada para componer los ficheros ejecutables ("linker") que monta ese ejecutable a partir de varios ficheros objeto, resolviendo sus interdependencias durante el montaje, y en el cargador de programas (un componente del sistema operativo que lee los ficheros ejecutables y deja algunas de sus regiones en memoria principal). Por tanto, si pretendemos aprovechar esos principios en el despliegue de aplicaciones distribuidas tendremos que aplicar algunos cambios en las herramientas que "cargan" y mantienen el código de los componentes y en las herramientas que gestionan el enlace entre puntos de acceso y puntos de dependencia, pero no será necesario aplicar ningún cambio en los programas de los componentes.

De esta manera, necesitaremos dos mecanismos complementarios: los contenedores y la inyección de dependencias.

4.1 Contenedores

Los contenedores son herramientas que mantienen a otros componentes *software*, proporcionándoles un entorno aislado (es decir, protegido) y la posibilidad de establecer una correspondencia entre sus puntos de acceso y los puntos de acceso del ordenador anfitrión. Además de esas correspondencias y la gestión del aislamiento, los contenedores gestionan las etapas del ciclo de vida de los componentes instalados en ellos, siendo capaces de generar (atendiendo las órdenes de sus usuarios) estos eventos relacionados con el ciclo de vida:

- **Creación**. Se necesitarán algunas órdenes para construir una imagen válida del componente a instalar en el contenedor. Para ello, algunos elementos dependientes del contenedor deben combinarse con el código del componente para generar esa imagen. Además, la imagen resultante también tendrá ciertos parámetros de configuración.
- **Registro** ("initiate"). Una vez se haya creado la imagen, tendrá que registrarse en el sistema de contenedores. Ese es el objetivo de este evento.
- **Inicio** ("start"). Este evento inicia la ejecución del componente en algún contenedor anfitrión.
- **Parada**. Este evento para la ejecución del componente. Posteriormente, la ejecución podrá ser reanudada o finalizada.
- **Reconfiguración**. Partiendo de alguna imagen previa, el evento de reconfiguración modifica el contenido de esta imagen (añadiendo o eliminando módulos en o de la imagen) o su configuración.
- **Destrucción**. Este evento elimina la imagen del componente del sistema de contenedores.

Un sistema de gestión de contenedores (como, por ejemplo, Docker) puede ser considerado una versión "ligera" de un hipervisor. Los sistemas hipervisores gestionan máquinas virtuales sobre un ordenador. Una imagen de máquina virtual comprende una pila completa de *software* (incluyendo el núcleo completo de un sistema operativo) que será ejecutada sobre un equipo virtualizado. Por su parte, las imágenes a utilizar en contenedores solo contienen el programa a ejecutar junto a un reducido conjunto de bibliotecas dependientes. Estas imágenes no necesitan incluir un núcleo de sistema operativo. El núcleo del sistema operativo anfitrión será compartido por todos los contenedores en ejecución en ese ordenador.

Por ejemplo, en Docker se necesita alguna distribución reciente de Linux como sistema operativo anfitrión. Las imágenes a ejecutar en Docker deben mantener algunas bibliotecas específicas de las que dependa el programa a ejecutar (llamemos P1 a ese programa). Con ello se crea una imagen de contenedor (a la que llamaremos C1). El resultado podrá ejecutarse sobre diferentes anfitriones Linux con configuraciones diversas. Las dependencias específicas de P1 estarán ya resueltas en la imagen C1. Por ejemplo, si P1 fuera un programa escrito en node que necesitara la versión 4.4.0 del intérprete de node, la imagen C1 contendría también ese intérprete y las bibliotecas que necesite para funcionar. La imagen C1 podrá ejecutarse sobre ordenadores anfitrión (con el gestor Docker instalado) que no necesitarán tener ninguna versión de node instalada o que podrían tener instalada cualquier otra versión del intérprete (p.ej., la 0.12.12). El aislamiento proporcionado por los contenedores soporta un escenario como este.

Los contenedores también facilitan cierta ayuda para gestionar el enlace dinámico de dependencias. El desarrollador de un componente puede escribir el código de su programa sin preocuparse por los puntos de acceso reales que se utilizarán. En lugar de ello, el programa asumirá algunas direcciones estáticas para sus puntos de acceso y sus puntos de dependencia. Durante la etapa de despliegue, el gestor de contenedores asociará esos puntos de entrada estáticos de la imagen en puntos de entrada reales en el ordenador anfitrión. Estos últimos serán los que deberán conocer y utilizar los demás componentes para interactuar con el componente que ahora desplegamos. Esto sería suficiente para resolver las dependencias cuando se utilicen *sockets* como mecanismo de intercomunicación, en los que los puntos de entrada constan de una dirección IP y un número de puerto.

4.2 Inyección de dependencias

Sin contenedores, cada componente cliente C estaría obligado a leer algún fichero de configuración para resolver sus dependencias respecto a otro componente servidor S. Eso implicaría una pérdida de transparencia, ya que el programa C sería consciente del mecanismo de resolución de dependencias utilizado (es decir, la lectura de un fichero de configuración y la interpretación adecuada de su contenido).

En lugar de leer un fichero de configuración, cuando se utilice un patrón de *inyección de dependencias* [11], el componente C solo necesitará conocer la interfaz de S para interactuar con él. Esa interfaz será implantada por alguna clase *proxy*. El objeto proxy P a utilizar para interactuar con S es "inyectado" en C cuando se realiza el despliegue (por ejemplo, tomando el código de P como uno de los elementos a incluir para construir la imagen del componente C). Así, si el punto de acceso a S cambiara, el programa C no necesitaría ser modificado. En su lugar, solo se necesitaría una nueva versión de P. Este patrón mantiene la transparencia propia de los

mecanismos de enlace dinámico. Los detalles de la resolución de dependencias están considerados en la implantación de P y tanto los programas de C como de S no deben sufrir modificaciones para interactuar correctamente.

5 COMPUTACIÓN EN LA NUBE

En las secciones anteriores se ha explicado qué es el despliegue, qué etapas podemos distinguir en él y qué métodos de configuración se necesitan para desplegar una aplicación distribuida. Todavía queda una cuestión pendiente... ¿dónde se desplegarán los servicios?

Tradicionalmente, los servicios de computación escalables han sido proporcionados por grandes empresas. Para ello, estas empresas mantenían grandes centros de computación, pero eso requería grandes inversiones. Esos centros de datos implicaban un alto coste de adquisición de equipos (ordenadores y redes) y también un alto coste operativo para mantener su servicio (considerando tanto su consumo energético como los salarios de los administradores como las renovaciones periódicas de los equipos).

Posteriormente, algunas empresas especializadas en el despliegue han ido apareciendo, alquilando su infraestructura a esas empresas proveedoras de servicios. Su negocio consiste en mantener grandes centros de datos, manteniendo las aplicaciones desplegadas por los proveedores de servicios. Así, estas empresas "anfitrionas" se han especializado en la administración de la infraestructura, reduciendo los costes operativos para los proveedores de servicios. Sin embargo, las primeras empresas de este tipo no disponían de grandes infraestructuras. Mantenían unos pocos centros de datos (en muchos casos, uno solo), alquilándolos a empresas proveedoras de servicios que ya estaban ubicadas cerca de ellas. Se necesitaban mejores soluciones: la gestión de esta infraestructura todavía debía ser más barata, más fiable y con mayor capacidad para la escalabilidad de los servicios desplegados en ella.

La solución a estos requisitos ha sido la computación en la nube. En concreto, el modelo de servicio IaaS ("Infrastructure as a Service"). En este modelo:

- El cliente solo paga por aquello que use; es decir, los costes dependerán del número de máquinas alquiladas y el ancho de banda utilizado.
- La unidad a utilizar es la máquina virtual (MV). Cada proveedor ofrece un conjunto de tipos de máquina virtual, según su memoria disponible y su capacidad de cómputo.
- El proveedor mantiene múltiples centros de datos dispersos geográficamente. Esto permite que los servicios se desplieguen cerca de sus usuarios potenciales.
- Se facilitan algunas herramientas para gestionar las MV alquiladas y los datos persistentes.

Pero este modelo de servicio todavía es demasiado primitivo por lo que respecta a la gestión de las tareas de despliegue porque...:

- Su interfaz permite seleccionar MV e instalar imágenes en ellas. Pero no hay reglas que automaticen las decisiones de escalado de los componentes desplegados. Para escalar hay que dar órdenes explícitas de asignación y liberación de máquinas virtuales y eso no

permite un escalado adaptativo (es decir, elástico). Los servicios elásticos deberían adaptarse por sí mismos a las variaciones de carga.

- Es responsabilidad del cliente la decisión sobre qué tipo de MV asignar en cada solicitud. Por tanto, el cliente debería ser un experto sobre las capacidades de las MV y las necesidades de los componentes a desplegar para tomar decisiones correctas.
- En muchos casos, el proveedor IaaS no facilita ninguna herramienta para monitorizar el tráfico en la red y seleccionar la ubicación de cada MV. Por tanto, resulta difícil resolver los problemas generados por una configuración de red inadecuada.
- Si la ubicación de las MV utilizadas no puede ser especificada, será difícil garantizar que no haya correlación entre los fallos que se den en las diferentes instancias de un componente desplegado (por ejemplo, podría darse el caso de que todas residieran en un mismo "rack" de un centro de datos y que todas quedaran inaccesibles si la subred que las interconecta fallara).

Por tanto, sería recomendable que hubiera mayores ayudas para automatizar el despliegue. Esas ayudas se proporcionan en el modelo de servicio PaaS ("Platform as a Service"). Este modelo de servicio tiene como objetivo automatizar las tareas del ciclo de vida de los servicios, incluyendo tanto el despliegue como la gestión de la elasticidad.

La clave para esa automatización es el SLA. En el SLA, cliente y proveedor firman un contrato sobre las propiedades no funcionales del servicio (principalmente, rendimiento y disponibilidad). A partir del SLA, el proveedor PaaS rellenará el plan de despliegue y establecerá las reglas de escalabilidad a utilizar para obtener una adaptabilidad óptima. Además, debería gestionar las actualizaciones del *software* de servicio respetando también el SLA durante esos intervalos.

Desafortunadamente, ninguno de los proveedores PaaS actuales ha alcanzado por completo esos objetivos, todavía. De momento, han conseguido automatizar parcialmente el despliegue inicial de los servicios (por ejemplo, el soporte para inyección de dependencias es todavía primitivo en algunos casos) pero todavía no automatizan las decisiones de escalado con suficiente precisión ni proporcionan procedimientos de actualización suficientemente fiables para evolucionar servicios "stateful". Esto se debe a que la gestión de los SLA es todavía insuficiente en esos sistemas PaaS.

Microsoft Azure (`azure.microsoft.com`) es uno de los sistemas PaaS actuales más evolucionados. Puede obtenerse más información sobre él en su sitio web oficial.

6 DOCKER

6.1 Introducción

En los apartados anteriores hemos estudiado:

- Un modelo de aplicación distribuida integrada por componentes autónomos
- Múltiples aspectos a contemplar en el despliegue, destacando:
  - La especificación y configuración de los componentes
  - El descriptor de despliegue
- Un ejemplo de despliegue (que conviene repasar)
- La problemática del destino del despliegue, introduciendo conceptos y modalidades relacionados con la Computación en la Nube (CC)

Este apartado pretende afianzar estos conceptos mediante un caso de despliegue, con suficientes detalles para realizarlo en el laboratorio. Las tecnologías concretas que intervienen son:

- Aplicaciones (en NodeJS) que se comunican mediante mensajería (ØMQ)
- Otro software adicional que puedan requerir las aplicaciones
- Sistemas (LINUX) sobre los que se ejecutan las aplicaciones con sus requisitos

Estas piezas (aplicación + requisitos + sistema) se reúnen formando un componente.

La tecnología que pretendemos emplear ha alcanzado un punto en el que existe un consenso acerca de su utilidad y aplicabilidad, pero las implementaciones que encontramos pueden ser inestables dada la velocidad de evolución actual. Es especialmente necesario señalar que la documentación, características y ejemplos pueden estar muy ligados a la versión de la implementación.

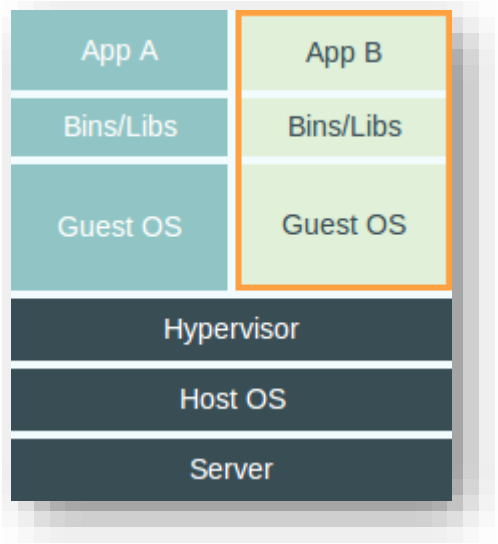
En el caso de Docker, que es una implementación de la tecnología de contenedores, hemos intentado que todo el material e informaciones recogidas sean compatibles con la versión *comunitaria* oficial al comenzar este curso 2023/24 (**docker-24.0.6**).

Es posible encontrar documentación y ejemplos anteriores a esta versión. También es posible confundirse con la herramienta docker-compose, que ha pasado por múltiples versiones y posibilidades. Es difícil detallar con antelación todos los factores que intervienen para reproducir el mismo escenario que se usará en los ejemplos y en el laboratorio, por lo que os animamos a manteneros alerta ante posibles errores e inexactitudes.

6.1.1 Aprovisionamiento

Llamamos **aprovisionamiento** (*provisioning*) a la tarea de reservar la infraestructura necesaria para que una aplicación distribuida pueda funcionar.

- Reservar recursos específicos para cada instancia de componente (procesador + memoria + almacenamiento).
- Reservar recursos para la intercomunicación entre componentes.

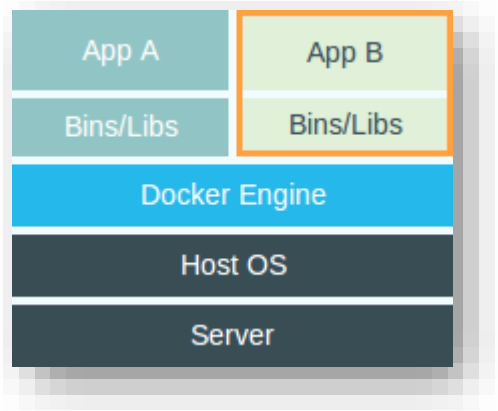


La infraestructura suele concretarse en un *pool* de **máquinas virtuales** interconectadas.

- El componente y sus requisitos se implementan y ejecutan sobre una máquina virtual.

En nuestro caso optamos por una versión *ligera*[1] de máquina virtual, denominada **contenedor.**

- El sistema del huésped coincide con el del anfitrión, por lo que no se requiere duplicarlo ni emularlo.
- El anfitrión ha de disponer de un software de *contenerización*, que ofrece ciertos servicios de aislamiento y replicación.



Cada componente se implementa sobre un contenedor.

6.1.2 Configuración de componentes

Para cada componente hay una especificación de su configuración que incluye...

- El software a ejecutar.
- Las dependencias que deberán ser concretadas.

Nuestra decisión anterior acerca de los contenedores supone que el software del componente…

- Debe ser compatible con el S.O. del anfitrión.
- Puede completarse con algún software adicional.
- Debe configurarse e inicializarse.

Estas tareas serán responsabilidad de la propia operación de creación de la imagen, y del fichero de propiedades del contenedor.

6.1.3 Plan de despliegue

Tal y como se describió en el apartado 3, la lista de acciones a ejecutar para llevar a cabo el despliegue viene especificada como un algoritmo o plan.

- En caso de una herramienta automatizada, existirá una aplicación que interpretará (*orquestación*) la especificación, y llevará a cabo las acciones.

[1] Ver segunda ilustración

- En su ausencia, se desarrollará un programa a medida de las indicaciones del plan de despliegue.

El despliegue manual se reserva para pruebas y casos sencillos, porque es inmanejable para una aplicación distribuida de tamaño medio.

6.2 De las máquinas virtuales a los contenedores

Actualmente las tecnologías de virtualización han permitido implementar servicios con flexibilidad, pero éstos deben rentabilizar una instalación completa. P.ej., una instalación virtualizada mínima puede consumir 500MB

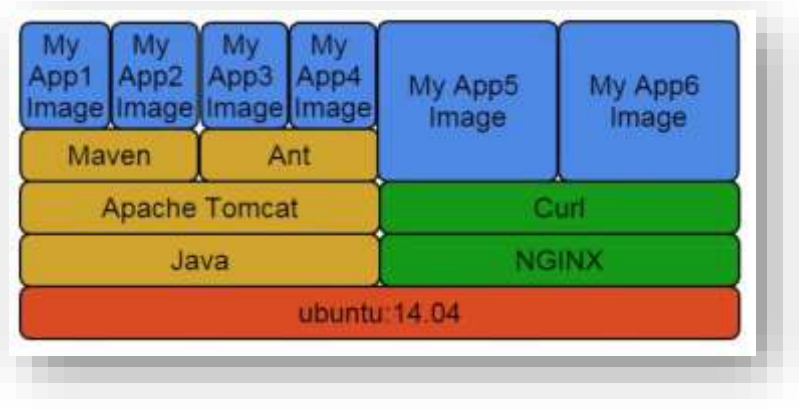
- No es práctico tomarla como base para un servicio que devuelva la hora actual (*demasiado equipaje*)
- Sí que es rentable como base para un servicio de correo electrónico con una decena de usuarios; por tanto, es adecuada para servicios con componentes relativamente pesados

Las implementaciones actuales en tecnología de contenedores, como Docker, son suficientemente maduras para plantearse como alternativa a las máquinas virtuales

- Ventaja: suponiendo que una imagen de 1GB usada por 100MVs consuma 100GBs…
  - Si hay 900MBs inmutables, 100 contenedores Docker consumirían 0,9+0,1*100=10,9GBs -> reduce **espacio**
    - Los contenedores consumen aproximadamente entre 10 y 100 veces menos recursos que sus equivalentes virtuales
  - Si la parte inmutable se encuentra "precargada", nos ahorramos ese tiempo (90%) para iniciar cada instancia Docker -> reduce **tiempo**

- Inconvenientes (como cualquier sistema de contenedores)
  - Menos flexible que las MVs
  - El aislamiento imperfecto entre contenedores puede provocar interferencias y problemas de seguridad

Con algunas simplificaciones, el sistema basado en contenedores ilustrado a la derecha nos permite detallar el ahorro respecto a un sistema basado en máquinas virtuales: el equivalente virtualizado daría lugar a **una MV completa por cada imagen de aplicación** (6 en total), incluyendo (de izquierda a derecha)…



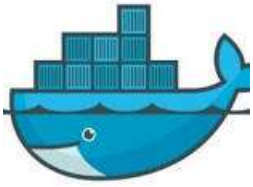
1. Ubuntu+Java+Tomcat+Maven+My App1 Image

2. Ubuntu+Java+Tomcat+Maven+My App2 Image
3. Ubuntu+Java+Tomcat+Ant+My App3 Image
4. Ubuntu+Java+Tomcat+Ant+My App4 Image
5. Ubuntu+NGINX+Curl+My App5 Image
6. Ubuntu+NGINX+Curl+My App6 Image

Sumando: 6*Ubuntu+4*Java+4*Tomcat+2*Maven+2*Ant+2*NGINX+2*Curl+My App1 Image+...+My App6 Image … **¡La diferencia es MUY sustancial!**

`La tecnología de contenedores es tan ligera que posibilita la virtualización` **`de una aplicación`**

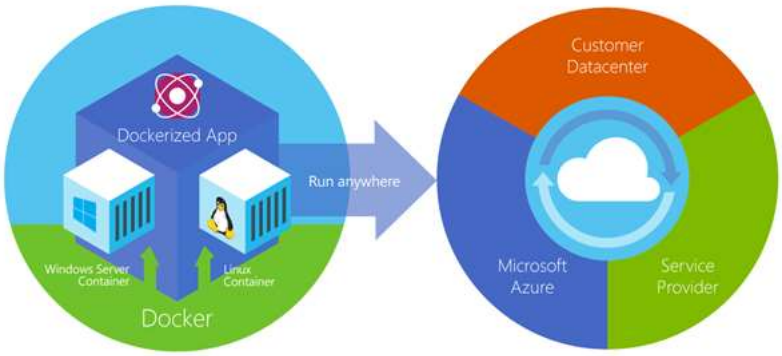
6.3 Introducción a Docker



Docker ofrece una API para ejecutar procesos de forma aislada, por lo puede tomarse como base para construir una PaaS

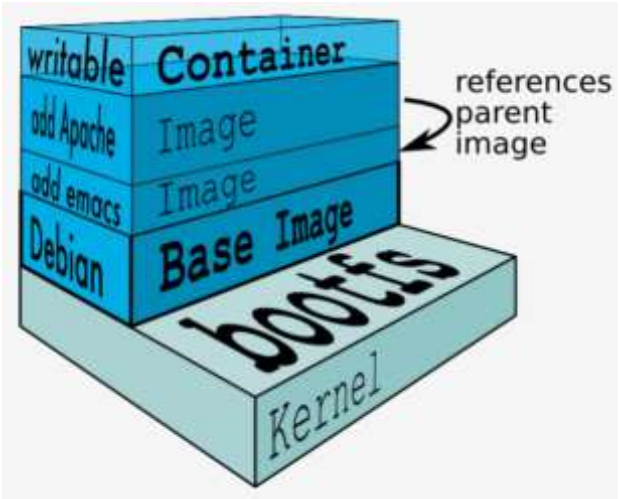
La implementación de Docker se basaba originalmente en LXC, pero posteriormente esa dependencia ha sido suavizada para interactuar con otros implementadores de contenedores. En una segunda etapa se utilizó *libcontainer* (ahora es parte de *runc*)

- Junto a Microsoft se ha conseguido una implementación nativa para Windows



Respecto a LXC, Docker añade:

- AuFS: sistema de ficheros con una parte de sólo lectura que puede compartirse
  - Las modificaciones forman capas que se superponen
  - Cada orden que cree, modifique o elimine uno o más ficheros generará una nueva capa en la imagen



En la página 28 puedes observar las capas de la imagen `misitio` (orden `docker history misitio`), construida con un Dockerfile para un servidor de web

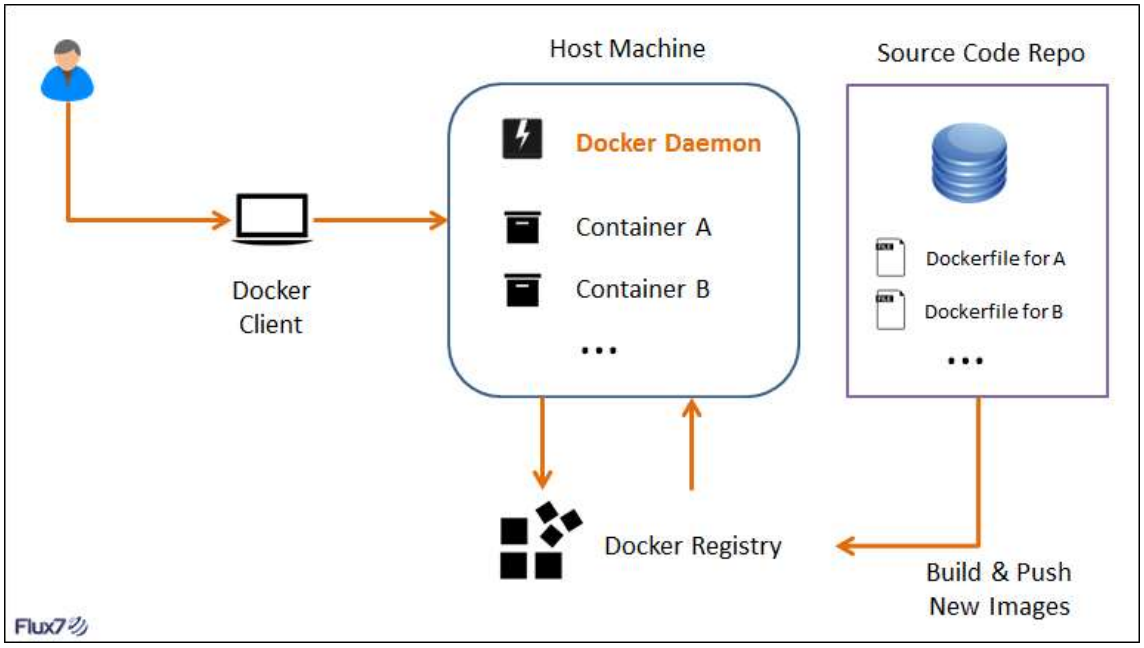
- Nueva funcionalidad:
  - construcción automática

- control de versiones (Git)
- compartición mediante depósitos públicos

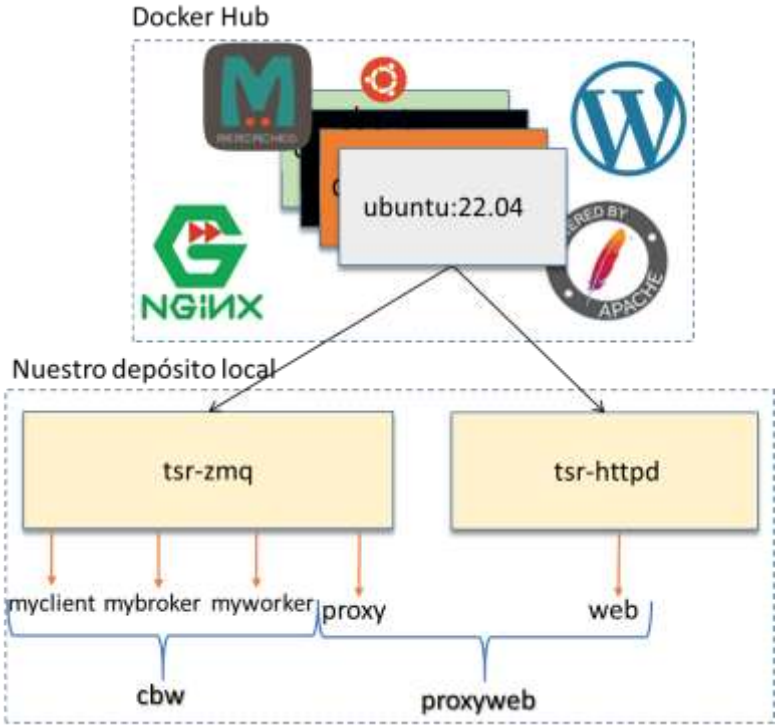
El sistema Docker dispone de 3 componentes:

1. **Imágenes** (componente *constructor*)
   - Las imágenes docker son básicamente plantillas de solo lectura a partir de las que se instancian contenedores
2. **Depósito** (componente *distribuidor*)
   - Hay un depósito común para poder subir y compartir imágenes (hub.docker.com)
3. **Contenedores** (componente *ejecutor*)
   - Se crean a partir de las imágenes, y contienen todo lo que nuestra aplicación necesita para ejecutarse
   - Se puede convertir un contenedor en imagen mediante docker commit

Internamente Docker consta de un *daemon*, un depósito y una aplicación cliente (docker)



A lo largo de este apartado emplearemos únicamente una imagen de depósito central, y derivaremos de ella algunas especializaciones para, al final, desplegar un par de aplicaciones multicomponente.



6.4 Funcionamiento

Tal como se comentará en el apartado 6.8, Docker está diseñado para ejecutarse organizando las actividades en un equipo individual (*standalone*), o como parte de un conjunto de nodos intercomunicados. En este último caso debe existir una interacción en la que participen, como un conjunto, los sistemas docker de cada nodo, existiendo dos alternativas:

- Una propia de Docker denominada **swarm** (enjambre). Si se emplea esta opción, cada sistema docker individual se ejecuta en *modo swarm*.
- Una alternativa externa a Docker, destacando la veterana Kubernetes (mayoritaria frente a Swarm).

La existencia del *modo swarm* supone que algunas órdenes de Docker solo tienen sentido en este modo de funcionamiento, y que la semántica de otras órdenes puede verse influenciada. En la práctica, en este documento, la descripción de las órdenes de Docker se basa en el modo *standalone*, evitando las funciones (grupos `node`, `stack` y `swarm`) relacionadas con el modo swarm.

Por otro lado, pese a su importancia, las operaciones relacionadas con la seguridad (autenticación mediante certificados, p.ej.) no son objeto de estudio en esta asignatura, lo que excluye los grupos de Docker relacionados (`secrets` y `trust`).

La sintaxis de Docker ha ido evolucionando para facilitar la incorporación de más de un centenar de órdenes. En su forma más actual, estas órdenes se agrupan atendiendo al objeto sobre el que actúan (p.ej, contenedores, imágenes, sistema, …). Así, para obtener una relación de las imágenes existentes se usaría `docker image list`. No obstante, es frecuente encontrar estas órdenes formuladas sin atender dichas agrupaciones, en un formato en el que no se indica expresamente el objeto sobre el que aplicar la orden, como `docker images`.

Las agrupaciones restantes tras la criba anterior son:

Grupo	Descripción
`config`	Gestión de configuraciones
`container`	Operaciones sobre contenedores
`context`	Contextos para el despliegue distribuido (k8s, …)
`image`	Gestión de imágenes
`network`	Gestión de redes
`service`	Gestión de servicios (contenedores instanciados desde la misma imagen) como parte de una aplicación distribuida (p.ej un SGBD)
`system`	Gestión global de Docker
`volume`	Gestión de almacenamiento persistente

6.4.1 Docker desde la línea de órdenes

El elemento crucial es el cliente docker mediante el que interactuamos con el demonio, lo que suele requerir privilegios para su uso.

```
docker [OPTIONS] COMMAND
```

Tipos de órdenes:

1. Control del ciclo de vida
   - docker **commit** (docker container commit)
   - docker **run** (docker container run)
   - docker **start** (docker container start)
   - docker **stop** (docker container stop)
2. Informativas
   - docker **logs** (docker container logs)
   - docker **ps** (docker service ps)
   - docker **info** (docker system info)
   - docker **images** (docker image ls)
   - docker **history** *imagen* (docker image history *imagen*)
3. Acceso al depósito
   - docker **pull** (docker image pull)
   - docker **push** (docker image push)
4. Varias
   - docker **cp** (docker container cp)
   - docker **export** (docker container export)

6.4.2 Ejemplo sencillo (orden `docker run`)

```
docker run -i -t imagen programa
```

- Acción: run sirve para construir y ejecutar (`-i -t` para interactivo)
  - P.ej. `docker run -i –t ubuntu /bin/bash`
- Imagen: p.ej ubuntu
- Programa: p.ej `/bin/bash`

Pasos:

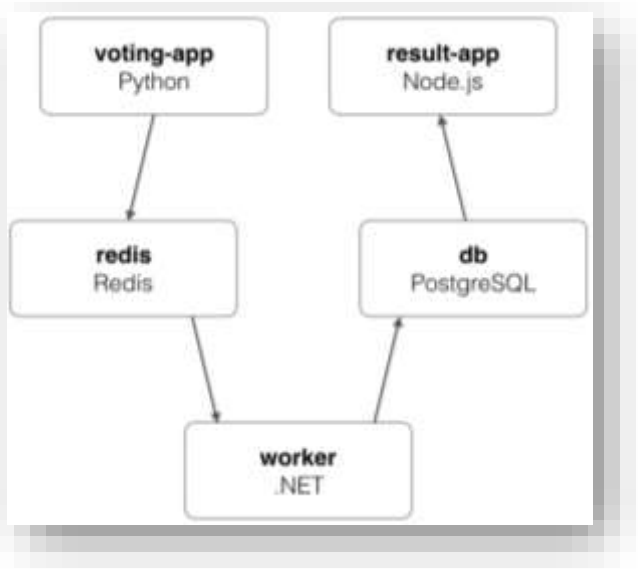
1. Descargar la imagen (ubuntu) desde el *Docker Hub*
2. Crear el contenedor.
3. Reservar un sistema de ficheros y añadir un nivel de lectura/escritura.
4. Reservar una interfaz de red para comunicar con el anfitrión.
5. Reservar una dirección IP interna.
6. Ejecutar el programa especificado (`/bin/bash`)
7. Capturar la salida de la aplicación

6.4.3 Algunos escenarios y ejemplos de uso

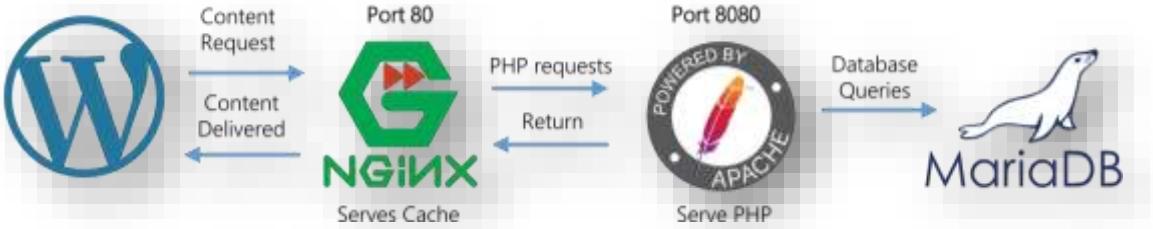
Hay una gran variedad de escenarios en los que se puede aplicar esta tecnología, y sería muy complejo establecer criterios para su clasificación; sin embargo es muy interesante disponer de **ejemplos interesantes e ilustrativos** que nos permitan entender el alcance de esta tecnología.

1. **Aplicaciones monolíticas**: aplicaciones de usuario (p.ej. Firefox, LibreOffice), aplicaciones servidor (p.ej. servidor de base de datos MySQL, servidor de web APACHE).
   - En estos casos la inclusión de todo un sistema operativo como requisito de la aplicación podría parecer desproporcionada, pero ahí reside la *magia* de la contenerización que consigue que el sistema sea rentable.

2. **Servicios multicomponente**: la propia aplicación se compone de varias piezas, con cierto grado de libertad, que se han de ensamblar para dar lugar al servicio final. En nuestro caso es interesante destacar estos tres ejemplos:

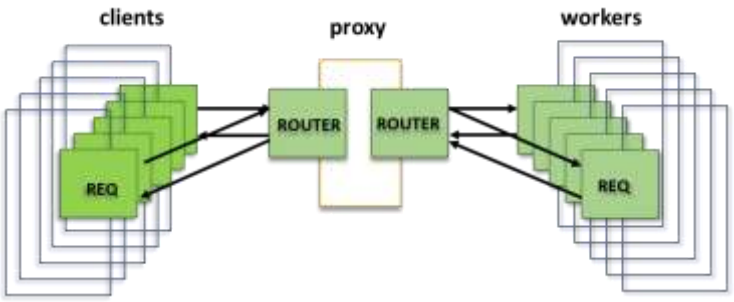


   - Un servicio de votaciones (*voting-app*[2]) que integra componentes desarrollados en sistemas muy diferentes (observa la ilustración a la derecha), pero que son capaces de interactuar una vez desplegados.
   - Un servicio de blog constituido por un proxy inverso Nginx, un servidor de web APACHE con soporte para aplicaciones PHP, el código de WordPress ejecutándose en el servidor, un servidor de base de datos MariaDB (compatible con MySQL).



     - Nuestro interés no se centra en el código fuente de las piezas sino en su configuración individual y sus relaciones con las demás, que serán los únicos aspectos en los que podamos intervenir.

   - Un servicio (p.ej. client+broker+worker) que nosotros mismos desarrollamos: no tendremos problemas en adaptarlo si fuera necesario, aunque esta capacidad no evita la preocupación por interconectar los diferentes componentes que participan de ese servicio.



[2] `https://github.com/dockersamples/example-voting-app`

3. **Despliegue de instalaciones genéricas**. Aunque pueda parecer lo más habitual, en el ámbito de la contenerización **no** suele abordarse el caso de equipos de escritorio completos, con todas las aplicaciones disponibles en una distribución tipo Ubuntu, Fedora o similares.
    - Es mucho más frecuente, y útil, encontrar instalaciones mínimas sobre las cuales se añaden las aplicaciones necesarias para cada caso. La contenerización apuesta por imágenes especializadas para cada caso. P.ej., una distribución mínima de Ubuntu sobre la que se ha añadido el soporte para ejecutar aplicaciones NodeJS.

Por otro lado, también se debe destacar que el **destino del despliegue** condiciona toda la operación, destacando dos extremos:

1. **Despliegue en un número discreto de equipos**. Se dispone de un conjunto de recursos limitado y poco flexible: como en el laboratorio de la asignatura. Los problemas a resolver deben encontrar soluciones compatibles con esta disponibilidad reducida, y esa falta de posibilidades reduce la complejidad del despliegue.
2. **Depliegue en la nube**. Existe un número elevado e indiferenciado de anfitriones. La cantidad de recursos disponibles es mucho mayor que en el caso anterior, y puede adaptarse *elásticamente* a las exigencias de cada momento. Infraestructuras de nube diferentes pueden imponer modos de despliegue concretos, o limitar nuestra libertad para especificar cómo se orquestan los componentes de la aplicación.

Por último, dentro de este apartado ilustrativo es importante mencionar los contenedores son efímeros, de manera se crea para ellos un sistema de ficheros que no sobrevive al contenedor: ¿de qué nos sirve realizar operaciones cuyo resultado se pierde al finalizar el contenedor?. La persistencia y otros problemas se resuelven mediante el **acceso a recursos del anfitrión**.

- Acceso al **sistema de ficheros**. Estableciendo una correspondencia entre el almacenamiento real del anfitrión y una ruta del sistema de ficheros efímero del contenedor, la información depositada no se destruye con dicho contenedor.
    - P.ej. usar la opción `-v` en `docker run`

```
docker run ... -v /ruta/anfitrion:/ruta/contenedor ...
```

    - Un caso especial es la posibilidad de interacción con el propio servicio Docker del anfitrión si el contenedor puede acceder al socket del servicio.
        - P.ej. usar la opción `-v` en `docker run`

```
docker run ... -v /var/run/docker.sock:/var/run/docker.sock ...
```

- Acceso a **puertos** del anfitrión. Estableciendo una correspondencia entre puertos del anfitrión y del contenedor, éste puede ser accedido desde el exterior.
    - P.ej. usar la opción `-p` en `docker run`

```
docker run ... - p 9999:9000 ...
```

    - Donde `9999` es el puerto del anfitrión que se hace corresponder con el puerto `9000` del contenedor.
    - También debe configurarse correctamente el cortafuegos del anfitrión, que puede limitar la visibilidad externa de esos puertos.

6.4.4 Servidor web mínimo sobre Ubuntu

Nota: Las actividades aplicadas, como ésta, únicamente pueden ser comprobadas en nuestros equipos virtuales de portal-ng

En este apartado pretendemos tomar contacto con un despliegue artesano, realizado de manera progresiva. El servicio a desplegar, al finalizar, consistirá en un servidor de web conectado al puerto 80 del contenedor que devuelve al navegador (`http://localhost/`) una página de bienvenida:

Este primer caso se ha ideado para emplearse en el laboratorio; nos sirve para poner a prueba nuestra infraestructura: software, configuración, puertos, etc… Los pasos[3] que realizamos para lograr este objetivo requieren previamente que averigüemos qué necesitamos para construir el servicio.

Abreviando, nuestra lista de requisitos es:

1. Una instalación base compatible con un servidor de web (p.ej. Ubuntu y el servidor APACHE). Es necesario prever cómo se consigue instalar el servidor sobre la imagen Ubuntu.
2. Material para *poblar* el directorio con documentos del servidor (p.ej. el contenido de `misitio.tar.gz`, que se encuentra en el directorio de este tema en PoliformaT[4]). Incluye carpetas con páginas, estilos, etc. Es necesario conocer dónde debe colocarse cada una de las piezas.
3. Vía de acceso al servicio resultante. Si ponemos en funcionamiento un navegador en la máquina virtual, ¿con qué URL accedemos al servicio del contenedor?. Debemos indicar en el anfitrión (nuestra virtual) que uno de sus puertos (p.ej el 8000) se asocie a un puerto (el 80) del contenedor.

[3] basado en `http://linoxide.com/linux-how-to/interactively-create-docker-container/`

Vayamos por pasos[5], pero sin perder la perspectiva global.

1. Conectamos con nuestra máquina virtual que dispone del software Docker necesario, y de una conexión a Internet.
2. Ejecutamos:

```
docker run -i -t ubuntu:22.04 bash
```

   - En esta instrucción seleccionamos la ejecución de una imagen llamada "ubuntu[6]".
   - Como Docker no dispone localmente de ninguna imagen con tal identificador, conectará con el depósito central (*docker hub*), encontrará esa imagen[7] y la traerá (72.8MBs).
3. Cuando consiga ponerlo en marcha, ejecutará en el contenedor el programa bash (un shell) y quedará a la espera de nuestras órdenes desde el teclado.
4. Usamos dentro del contenedor la orden apt-get (¡ya estamos en Ubuntu!) que es un gestor de paquetes.
   - Primero deberíamos ponerlo ponerlo al día (`apt-get update -y; apt-get upgrade -y`),
   - y después añadimos el servidor APACHE (paquete `apache2`). Esta actualización necesita transferir cerca de 57 paquetes nuevos (unos 24MBs).

```
(dentro del contenedor) apt-get install -y apache2
```

(Seguramente deberemos proporcionar algo más de información en esta instalación, como la ubicación geográfica para conocer el huso horario)

Una enorme ventaja de Docker es que mantiene una copia local de aquello que pueda reutilizar, de modo que una nueva imagen basada en otra usada anteriormente, requeriría poco esfuerzo.

5. De momento no necesitamos nada más dentro del contenedor, así que salimos de él y volvemos a nuestra virtual.

```
(dentro del contenedor) exit
```

   - En líneas generales, todo lo que hacemos con los contenedores es volátil salvo que digamos lo contrario.

6. Debemos averiguar el nombre interno del contenedor. Como ya hemos terminado con él, buscaremos en el sistema un contenedor basado en ubuntu:22.04 que haya finalizado recientemente (unos minutos a lo sumo)

```
docker ps -a
// elegimos el más reciente basado en ubuntu:22.04
// copiamos los primeros dígitos de la columna CONTAINER_ID
// y hacemos un commit para dar nombre y congelar la imagen
```

[5] 12 en total

[6] Se trata de una distribución de LINUX emparentada con Debian, versión Focal Fossa

[7] De hecho es una familia en la que cada miembro ubuntu tiene asociado un número de versión. Si se omite, se traerá el último (latest).

```
docker commit CONTAINER_ID tsr-httpd
```

7. Como resultado, si ejecutamos docker images observaremos dos elementos: la imagen ubuntu descargada del *Docker Hub*, y nuestra ubuntu-httpd que acabamos de crear localmente.

```
docker images
```

- Con esto terminamos con el primer requisito y pasamos al segundo (poblar directorio).

8. Copiamos `misitio.tar.gz` a nuestra virtual y lo descomprimimos, dando lugar a una carpeta misitio. Ahora hemos de instruir al contenedor para que "coja" esos materiales. Recurriremos a operaciones automatizadas a través del archivo de configuración Dockerfile[8] cuyo contenido será:

```
FROM tsr-httpd
ADD misitio/index.html /var/www/html/index.html
ADD misitio/css /var/www/html/css
ADD misitio/js /var/www/html/js
ADD misitio/fonts /var/www/html/fonts
EXPOSE 80
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

9. En ese mismo directorio ejecutamos:

```
docker build --rm -t misitio .
```

- ¡No olvides el punto al final de la orden!

Con esto instruimos a Docker para que a partir del Dockerfile del directorio actual genere un contenedor llamado `misitio` *eliminando los niveles intermedios*[9] que pueda ir generando.

```
Sending build context to Docker daemon  1.373MB
Step 1/7 : FROM tsr-httpd
 ---> 53aa1ec95dc3
Step 2/7 : ADD misitio/index.html /var/www/html/index.html
 ---> 2df9e0613c0e
Step 3/7 : ADD misitio/css /var/www/html/css
 ---> 380159e679b7
Step 4/7 : ADD misitio/js /var/www/html/js
 ---> 359149818622
Step 5/7 : ADD misitio/fonts /var/www/html/fonts
 ---> 65292d603dae
Step 6/7 : EXPOSE 80
 ---> Running in d458f035fae7
Removing intermediate container d458f035fae7
 ---> 5ff52acdffbf
Step 7/7 : CMD /usr/sbin/apache2ctl -D FOREGROUND
 ---> Running in 3baaeaca8452
Removing intermediate container 3baaeaca8452
 ---> 3b0904998b80
Successfully built 3b0904998b80
Successfully tagged misitio:latest
```

[8] Dockerfile es objeto de estudio en uno de los próximos apartados
[9] Cada nivel se obtiene como resultado de ejecutar una orden del Dockerfile

A título informativo, la imagen `misitio` se compone de las siguientes capas[10] (`docker history misitio`):

```
IMAGE          CREATED              CREATED BY                                      SIZE
3b0904998b80   About a minute ago   /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "/usr…    0B
5ff52acdffbf   About a minute ago   /bin/sh -c #(nop) EXPOSE 80                     0B
65292d603dae   About a minute ago   /bin/sh -c #(nop) ADD dir:34c0d5e8b4ecf03f6c…   216kB
359149818622   About a minute ago   /bin/sh -c #(nop) ADD dir:2b98d70a6d3ddcdea2…   106kB
380159e679b7   About a minute ago   /bin/sh -c #(nop) ADD dir:75ba790fa14fc0c7c1…   760kB
2df9e0613c0e   About a minute ago   /bin/sh -c #(nop) ADD file:97e2afd872c91d3cd…   924B
53aa1ec95dc3   7 minutes ago        bash                                            162MB
a0ce5a295b63   2 months ago         /bin/sh -c #(nop) CMD ["bash"]                  0B
<missing>      2 months ago         /bin/sh -c #(nop) ADD file:ff6963f777661fb16…   72.8MB
```

Pero la opción `--rm` provoca que las imágenes intermedias no se guarden.

10. Esto finaliza el segundo requisito, y nos quedará el tercero: ejecutar el contenedor permitiendo el acceso al servicio que ofrece.

```
docker run -p 8000:80 -d -P misitio
```

- Esta orden pone en funcionamiento el contenedor, asociando el puerto 8000 de la virtual como punto de entrada para el 80 del contenedor.

11. Con un navegador en la máquina virtual que acceda al url `http://localhost:8000` deberíamos ver la ilustración que aparece al comienzo de este subapartado.
    - Cuestión: ¿Qué necesitarías hacer para acceder al servidor desde tu sesión de escritorio?

12. Para terminar con el contenedor, averiguamos su identificador con docker ps[11], y lo detenemos con docker stop.
    - La prueba de fuego consiste en recargar el navegador, que fallará al no poder conectar.

Conclusiones destacables

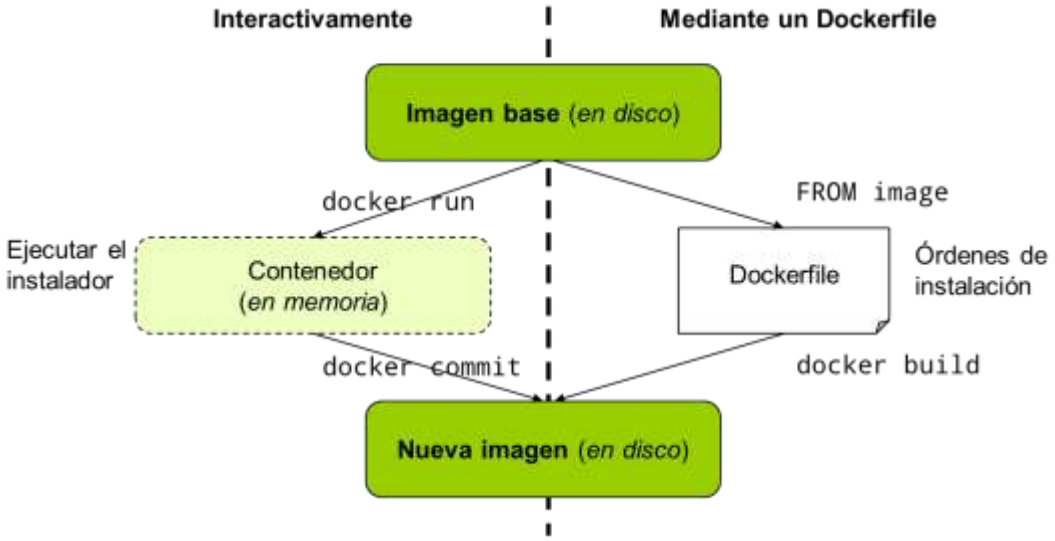
- La creación de una imagen nueva puede consumir tiempo y recursos de forma apreciable. Si la imagen se basa en otra que ya tenemos, el sistema reduce los pasos necesarios y el consumo de recursos. Ésta es una de las razones que aconsejan reaprovechar las imágenes ya creadas.
- Un Dockerfile completo permite reproducir las acciones con comodidad y reduciendo el riesgo de errores, pero es difícil escribirlo correctamente a la primera. El método de prueba y error mediante una sesión interactiva es una buena aproximación. Consultar la documentación es **imprescindible**.

Preparar un contenedor para ejecutar aplicaciones NodeJS con ZeroMQ

Podemos proceder a construir una imagen de forma interactiva y progresiva (*prueba y error*) mediante algún *shell* sobre el contenedor: iríamos anotando los pasos que nos parezcan

[10] Referenciado desde la página 18 al hablar de las capas de una imagen
[11] No necesita opciones porque el contenedor se encuentra en ejecución

apropiados. Más tarde, con esa lista, podremos automatizar la generación de la imagen en un Dockerfile.



Aunque el número y variedad de imágenes disponibles, producidas por otra gente, es muy elevado, no suele encontrarse ninguna que se ajuste exactamente a nuestras necesidades. Sin embargo, sí que es muy habitual que varias de ellas puedan ser tomadas como referencia para añadir posteriormente nuestro middleware y nuestra aplicación final.

En este apartado nos marcamos como objetivo construir un contenedor en el que se pueda ejecutar una aplicación NodeJS con la biblioteca ZMQ. Por familiaridad, tomaremos como base una distribución Ubuntu[12] Jammy Jellyfish, ó 22.04, de la que nos interesa especialmente la gestión de paquetes mediante apt-get: podemos instalar aplicaciones de manera que sus prerrequisitos se instalen automáticamente por la red.

En detalle. los pasos a seguir son...

1. Averiguar los **prerrequisitos** para realizar dicha instalación sobre una distribución Ubuntu:
   - Siempre el punto de partida supone actualizar la información sobre los diferentes depósitos de los que proceden las aplicaciones y bibliotecas necesarias.

```
apt-get update -y
```

   - Leyendo documentación observamos que no vienen preinstalados algunos programas que necesitamos, especialmente curl y las herramientas de compilación. Los añadimos:

```
apt-get install curl ufw gcc g++ make gnupg -y
```

   - Necesitamos dar de alta el depósito nodesource para poder instalar NodeJS

[12] hay mucha información disponible en la red

Ejecutar el script[13] en https://deb.nodesource.com/setup_20.x para añadir este depósito, particularizado para la serie 20

```
curl –sL https://deb.nodesource.com/setup_20.x | bash -
```

- Actualizamos el sistema para que considere esta nueva fuente:

```
apt-get update -y
```

- Instalamos NodeJS y actualizamos todas las aplicaciones:

```
apt-get install nodejs -y; apt-get upgrade -y
```

- Mediante npm instalamos ZeroMQ y su enganche para NodeJS:

```
npm init -y; npm install zeromq@5
```

- Esta acción provoca un error que debe ser resuelto. Tirando de internet averiguamos que esta orden no funciona en el directorio raíz ("/"), por lo que deberemos ejecutarla dentro de un directorio, como `/root`

```
cd /root (antes de la orden anterior)
```

Atención: En la versión final colocaremos esta última instrucción como la primera.

2. Cuando ya tengamos claros esos requisitos, habrá que aplicar esas acciones para generar una nueva imagen. El primer paso será lanzar un contenedor para ejecutar la imagen Ubuntu en modo interactivo:

```
docker run –i –t ubuntu:22.04 bash
```

3. Desde el intérprete de órdenes de ese contenedor, iremos invocando las siguientes órdenes:

```
$ cd /root
$ apt-get update -y
$ apt-get install curl ufw gcc g++ make gnupg -y
$ curl -sL https://deb.nodesource.com/setup_20.x | bash -
$ apt-get update –y
$ apt-get install nodejs -y; apt-get upgrade -y
$ npm init –y; npm install zeromq@5
$ exit
```

4. Desde la línea de órdenes de nuestro sistema (anfitrión), posiblemente en otra ventana, obtenemos el identificador o nombre del contenedor utilizado en los pasos anteriores:

```
docker ps -a
CONTAINER ID   IMAGE          COMMAND   CREATED          STATUS          PORTS     NAMES
23cca85b56b3   ubuntu:22.04   "bash"    6 minutes ago    Up 6 minutes              recursing_nobel
```

5. Realizar el commit del contenido actual del contenedor, generando así una nueva imagen

```
docker commit recursing_nobel tsr-zmq
```

o, alternativamente, especificando los primeros símbolos del id de contenedor:

```
docker commit 23cca tsr-zmq
```

[13] Incluye una advertencia sobre la obsolescencia del script, y un retardo *desorientador* de 60 segundos

- Ahora ya tendremos una imagen Docker llamada "tsr-zmq" desde la que lanzar programas node en contenedores.

6. Comprobar mediante `docker images`

Estos pasos que hemos realizado pueden ser automatizados si se colocan las instrucciones y propiedades en un archivo de configuración (Dockerfile) que estudiamos en el siguiente apartado.

- La imagen anterior solo ha sido producida para ilustrar los pasos necesarios en su construcción. No la usaremos en el resto de este tema **y puede eliminarse**.

Aunque no tomemos esa alternativa, es posible identificar varias etapas en la creación de esa imagen. Por ejemplo:

- una primera que llegaría hasta la instalación de NodeJS,
- y otra segunda que añadiría ZeroMQ a la anterior.

6.5 Dockerfile: El fichero de propiedades de Docker

Docker puede **construir una imagen** a partir de las instrucciones de un fichero de texto llamado Dockerfile, que debe encontrarse en la raíz del depósito que deseemos construir

```
docker build <opciones> <ruta_depósito>
```

Sintaxis general: INSTRUCCIÓN argumento

- Por convención, las instrucciones se escriben en mayúsculas
- Se ejecutan por orden de aparición en el Dockerfile

Todo Dockerfile debe comenzar con FROM, que especifica la imagen que se toma como base para construir la nueva.

- Sintaxis: FROM <nombre_imagen>

6.5.1 Órdenes en el archivo Dockerfile

Dispones de material de referencia en la "**Docker reference documentation**" de este mismo tema.

1. **MAINTAINER**: Establece el autor de la imagen, pero actualmente se prefiere el uso de la orden `LABEL maintainer=<nombre autor>`
2. **RUN**: Ejecuta una orden (*shell* o *exec*), añadiendo un nuevo nivel sobre la imagen resultante. El resultado se toma como base para la siguiente instrucción
   - Sintaxis: `RUN <orden>`
3. **ADD**: Copia archivos de un lugar a otro
   - Sintaxis: `ADD <origen> <destino>`
   - El origen puede ser un URL, un directorio (se copia todo su contenido) o un archivo accesible en el contexto de esta ejecución
   - El destino es una ruta en el contenedor
4. **CMD**: Esta orden proporciona los valores por defecto en la ejecución del contenedor

Sólo puede usarse una vez (si hubiera varias, sólo se ejecutará la última)

- Sintaxis: 3 alternativas
  - `CMD ["ejecutable","param1","param2"]`
  - `CMD ["param1","param2"]`
  - `CMD orden param1 param2`

5. **EXPOSE**: Indica el puerto en el que el contenedor atenderá (*listen*) peticiones
    - Sintaxis: `EXPOSE <puerto>`
6. **ENTRYPOINT**: Configura un contenedor como si fuera un ejecutable
    - Especifica una aplicación que se ejecutará automáticamente cada vez que se instancie un contenedor a partir de esta imagen
        - Implica que éste será el único propósito de la imagen
    - Como en `CMD`, sólo se ejecutará el último `ENTRYPOINT` especificado
    - Sintaxis: 2 alternativas
        - `ENTRYPOINT ["ejecutable", "param1", "param2"]`
        - `ENTRYPOINT orden param1 param2`
7. **WORKDIR**: Establece el directorio de trabajo para las instrucciones `RUN`, `CMD` y `ENTRYPOINT`.
    - Sintaxis: `WORKDIR /ruta/a/directorio_de_trabajo`
8. **ENV**: Asigna valores a las variables de entorno que pueden ser consultadas por los programas dentro del contenedor.
    - Sintaxis: `ENV <variable> <valor>`
9. **USER**: Establece el UID bajo el que se ejecutará la imagen.
    - Sintaxis: `USER <uid>`
10. **VOLUME**: Permite el acceso del contenedor a un directorio del anfitrión, lo que permite que su contenido "*sobreviva*" a la ejecución del contenedor. Se usa en 2 momentos complementarios: primero se especifica la ruta deseada dentro del contenedor, y en la orden docker run se establece el equivalente en el anfitrión
    - Sintaxis: `VOLUME ["/ruta/contenedor"]` (en Dockerfile)
    - Sintaxis: `docker run ... -v /ruta/anfitrion:/ruta/contenedor ...` (en línea órdenes)

Aunque sean similares, algunas diferencias entre `RUN`, `CMD` y `ENTRYPOINT` pueden ser importantes y requieren estudio. Es fácil diferenciar `RUN`: se refiere a órdenes necesarias internamente para construir la imagen, pero se ejecutan en la construcción, no para cada invocación del contenedor instanciado.

- Cada orden `RUN` genera una nueva capa en la construcción de la imagen

Imagina[14] que se trata de una imagen (*example_container*) con un Dockerfile que simplemente instancia una imagen basada en LINUX y luego muestra un mensaje en pantalla.

Si el Dockerfile de *example_container* dispone de la línea `CMD`, entonces ésa es la orden por defecto a ejecutar.

```
FROM ubuntu:22.04
CMD ["/bin/echo", "Hello"]
```

- `docker run -it example_container`, sin parámetros adicionales, ejecutaría `/bin/echo "Hello"`
- `docker run -it example_container /bin/sh`, ejecutaría un *shell* en el contenedor (no se invocaría `/bin/echo "Hello"`)

[14] Ejemplo en https://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/

Si el Dockerfile de *example_container* dispone de la línea `ENTRYPOINT`, entonces no se puede proporcionar ningún argumento externo.

```
FROM ubuntu:22.04
ENTRYPOINT ["/bin/echo", "Hello"]
```

- `docker run -it example_container /bin/sh`, ignoraría el argumento (`/bin/sh`) y ejecutaría `/bin/echo "Hello"`

En cierto modo `ENTRYPOINT` es una versión restrictiva de `CMD`, pero se complementan si ambos aparecen en un mismo Dockerfile: se ejecuta la orden referenciada mediante `ENTRYPOINT` agregando a sus parámetros los argumentos de `CMD`

```
FROM ubuntu:22.04
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["World"]
```

- `docker run -it example_container`, sin parámetros adicionales, ejecutaría `/bin/echo "Hello World"`
- `docker run -it example_container Moon`, con ese parámetro adicional, ejecutaría `/bin/echo "Hello Moon"`

6.5.2 Ejemplos de Dockerfile

Puedes consultar otros ejemplos en la documentación adicional y en muchos lugares de la web, como https://hub.docker.com/u/komljen/

Ejemplo 1: client/broker/worker con ZeroMQ

Este ejemplo se introdujo en la práctica 2: un servicio implantado mediante componentes client, broker (tipo ROUTER-ROUTER) y worker, que podrá replicarse tantas veces como sea necesario.

Tanto el cliente como el trabajador necesitarán como primer argumento el URL del broker, pero el resto de informaciones se dejarán en blanco.

- Esto supone que no se negociarán los protocolos ni puertos de conexión.

Por comodidad se añade el código de estos 3 componentes (*myclient*, *mybroker*, *myworker*) en el primer apéndice de este documento.

Nos basamos en una imagen Docker llamada "`tsr-zmq`" con...

- NodeJS instalado sobre una imagen de alguna distribución[15] Linux.
- La biblioteca ZeroMQ instalada correctamente.
- El módulo zeromq instalado y disponible para su uso desde node.
- Sobre esta imagen se podrá ejecutar cualquier programa NodeJS que utilice ØMQ.

[15] Ubuntu en este caso

El Dockerfile (con comentarios) para crear dicha imagen "**tsr-zmq**" es:

```
# Take an LTS Ubuntu distribution as a base.
# For example Ubuntu 22.04 "Jammy Jellyfish"
FROM ubuntu:22.04
# Install the latest Node.js on that distribution.
# First of all, preparing initial directory and repo packages synchronization
WORKDIR /root
RUN apt-get update -y
 # Second: install prerrequisites
RUN apt-get install curl ufw gcc g++ make gnupg -y
# Next: install NodeJS v20 (three steps)
RUN curl -sL https://deb.nodesource.com/setup_20.x | bash -
RUN apt-get update -y
RUN apt-get install nodejs -y
 # Fourth: system up-to-date
RUN apt-get upgrade -y
# Last: initialize npm, add ZeroMQ for NodeJS apps
RUN npm init -y
RUN npm install zeromq@5
```

Pasos a seguir para utilizar este servicio...

1. Crear la imagen Docker para el componente **broker**.

Necesitamos un Dockerfile similar a éste:

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir broker
WORKDIR broker
COPY ./broker.js mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

- La orden COPY supone que el código del broker se encuentra en este mismo directorio, y lo copia a un directorio del contenedor que se está utilizando para construir la imagen.
- Se deja público el puerto 9998 para los clientes...
- ... y el 9999 para los "*workers*".

Y generamos su contenedor con esta orden:

```
docker build -t broker .
```

- Lanzamos el broker (`docker run --name mybroker broker`[16]) y averiguamos su IP mediante…

```
 docker inspect mybroker | grep IPAddress | cut -d '"' -f 4
```

- Anotamos la IP que aparece (p.ej. a.b.c.d) y la usamos para construir el Dockerfile de los otros 2 componentes

2. Crear la imagen Docker para el componente **worker**.

Necesitamos un Dockerfile similar a éste:

```
FROM tsr-zmq
```

[16] Con `--name` damos un identificador a esta instancia (contenedor `mybroker`) para poderlo usar como argumento de `docker inspect`

```
COPY ./tsr.js tsr.js
RUN mkdir worker
WORKDIR worker
COPY ./workerReq.js myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker a.b.c.d 9999
```

- Y generamos su imagen con esta orden estando en el directorio *worker*:

```
docker build –t worker .
```

3. Crear la imagen Docker para el componente **client**.

Necesitamos un Dockerfile similar a éste:

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir client
WORKDIR client
COPY ./cliente.js myclient.js
# We assume that each client is linked to the broker
# container.
CMD node myclient a.b.c.d 9998
```

- Y generamos su imagen desde el directorio *client* con esta orden docker:

```
docker build –t client .
```

Ahora ya podemos ejecutar estos procesos comprobando que conectan con el broker y que envían y reciben mensajes.

Ejemplo 2: TCPProxy para un servidor Web

Este ejemplo gira en torno a la adición de un componente (proxy TCP) a una instalación de un servidor de web monolítico extremadamente sencillo.

El código para el proxy TCP de la práctica 1 (ProxyConf.js) es un buen punto de partida para experimentar un nuevo despliegue de un servidor web, de manera que este proxy debe intermediar en las peticiones de servicio procedentes de los clientes, reenviándolas al puerto e IP del servidor web.

Deseamos crear el Dockerfile necesario para este componente, y, más tarde, un **docker-compose.yml** en el que se incorporen proxy y servidor web.

Es importante observar que la aplicación distribuida ha de mantener las mismas interfaces externas, de manera que el puerto 80, anteriormente asociado al servidor web, ahora deberá vincularse al proxy TCP. Esto NO obliga a cambiar el puerto del componente web, sino su visibilidad (p.ej. mediante expose) como acceso al servicio en el fichero de despliegue docker-compose.yml.

Dockerfile

Posee una parte general sobre la que tendremos que aplicar algunos ajustes:

```
FROM tsr-zmq[17]
```

[17] Aunque no necesitamos el soporte para ZMQ

a) El programa ***ProxyConf.js*** a ejecutar, que hay que copiar previamente.

```
COPY ./ProxyConf.js Proxy.js
```

b) Hacer accesible el puerto de servicio al que atenderá este proxy TCP

```
EXPOSE 80
```

c) Colocar los parámetros adecuados del componente **web** (IP y puerto) para que sean transmitidos al programa en su invocación.

```
CMD node Proxy IP? puerto?
```

Por tanto, deberíamos averiguar antes los detalles necesarios, editar el Dockerfile y ejecutar:

```
docker build -t proxy .
```

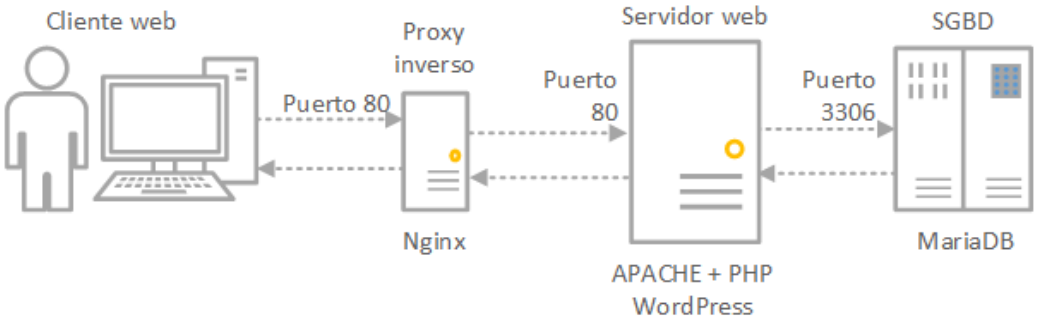
6.6 Múltiples componentes

En los ejemplos de servicios multicomponente del apartado anterior se facilita el escalado y la disponibilidad colocando cada componente en un contenedor. Tomemos como ejemplo el servicio de blog basado en WordPress, construido como:

- El servidor de web APACHE es el eje central del esquema.
  - Incluye un intérprete de PHP que ejecuta la aplicación WordPress para cada petición que le llega.
  - Habitualmente WordPress interactúa con el SGBD para leer y escribir información.
- Para aligerar el trabajo del servidor de web, hay un proxy inverso HTTP (Nginx) que filtra las peticiones sencillas y las atiende

Podemos ubicar el proxy inverso, el servidor de web y el SGBD en **contenedores intercomunicados**.

- WordPress requiere la presencia del intérprete de PHP y no puede separarse de APACHE[18]



El despliegue ahora se complica porque se encuentran dependencias que deben ser resueltas:

- ¿cómo sabe el primer componente (proxy inverso) los detalles necesarios para conectar con el segundo (servidor web)?
  - Al menos su IP (resto de valores por defecto)

[18] al menos con facilidad

- Puede que no se resuelvan hasta que el segundo componente inicie su ejecución
- La misma pregunta se puede realizar para el servidor de web y el SGBD

Simplificadamente, se requiere …

1. Crear los Dockerfile de los 3 componentes
    - Serán un subconjunto del mostrado anteriormente
2. Iniciar el tercero (SGBD), obteniendo su IP
3. Iniciar el segundo (servidor web), transmitiéndole la IP del tercero, obteniendo su IP
4. Iniciar el primero (proxy inverso), transmitiéndole la IP del segundo

Sin embargo, esta aproximación artesana es inaplicable para casos de envergadura media y grande. Se necesita:

- Lenguaje para generalizar la descripción de los despliegues
    - Que pueda expresar componentes, propiedades y relaciones
    - P.ej. YAML (para Compose, Kubernetes), OASIS-TOSCA
- Automatizar (***orquestar***) la ejecución de los despliegues
    - Mediante un motor que ejecuta el despliegue según la descripción
    - P.ej. Docker-Compose[19], Kubernetes[20], APACHE-Brooklyn

Es conveniente disponer de herramientas que faciliten la creación y simulación de estos despliegues, como https://lorry.io/

- ¡¡Los imprevistos no son bienvenidos mientras se realiza un despliegue de gran magnitud!!

Docker admite "enlaces" entre contenedores cuyo establecimiento puede **automatizarse mediante "docker-compose"**.

6.7 Despliegue en docker compose

`docker-compose` es una aplicación para definir y ejecutar aplicaciones ubicadas en varios contenedores Docker. La herramienta es un producto en desarrollo, por lo que muchos detalles de la misma cambian con facilidad en poco tiempo. Recientemente se ha producido una gran reescritura de la aplicación[21], y puede llegar a invocarse como si se tratara de una opción más de la orden `docker` (aunque no lo es).

Tres pasos:

1. Definir el entorno de la aplicación con un **Dockerfile**
2. Definir los servicios que constituyen la aplicación en un archivo `docker-compose.yml` para que puedan ejecutarse conjuntamente
3. Ejecutar `docker-compose up`, con lo que Compose iniciará y ejecutará la aplicación completa

[19] Solo si no afecta a más de un equipo
[20] https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes, https://kubernetes.io/
[21] https://www.docker.com/blog/announcing-compose-v2-general-availability/, abril de 2022

Limitado a contenedores en el mismo equipo, pero puede completarse con Docker Swarm (u otro software de orquestación) para controlar un *cluster*.

Esta aplicación interactúa con el *daemon* docker y no viene incluida en Docker.

6.7.1 Características destacables

- Se puede disponer de varios entornos aislados en un mismo equipo
- Se pueden ubicar los datos fuera de los contenedores (volúmenes)
- Sólo hay que crear de nuevo los contenedores que se hayan modificado
- En docker-compose.yml se pueden usar capacidades para adaptarse al entorno

El elemento crucial es el programa `docker-compose`

```
docker-compose [OPTIONS] [COMMAND] [ARGS...]
```

Ciclo típico de uso:

```
$ docker-compose up -d
… actividades …
$ docker-compose stop
$ docker-compose rm -f
```

6.7.2 Docker Compose desde la CLI

Órdenes más significativas:

- **build**: (re-)construye un servicio
- **kill**: detiene un contenedor
- **logs**: muestra la salida de los contenedores
- **port**: muestra el puerto asociado al servicio2
- **ps**: lista los contenedores
- **pull**: sube una imagen
- **rm**: elimina un contenedor detenido
- **run**: ejecuta una orden en un servicio
- **scale**: número de contenedores a ejecutar para un servicio
- **start** | **stop** | **restart**: inicia | detiene | reinicia un servicio
- **up**: build + start (*aproximadamente*)

Ejemplo de orden **run**

```
$ docker-compose run web python manage.py shell
```

1. Inicia el servicio **web**
2. Envía al servicio la orden **python manage.py shell** para que la ejecute
3. La ejecución de esa orden en el servicio se desvincula de nuestro shell (desde donde hemos lanzado docker-compose)

Hay dos diferencias entre **run** y **start**

- La orden que pasamos a run tiene preferencia sobre la que se haya especificado en el contenedor
- Si hay puertos en colisión con otros ya abiertos, no se crearán los puertos nuevos

6.7.3 El fichero de descripción del despliegue docker-compose.yml

https://docs.docker.com/compose/compose-file/

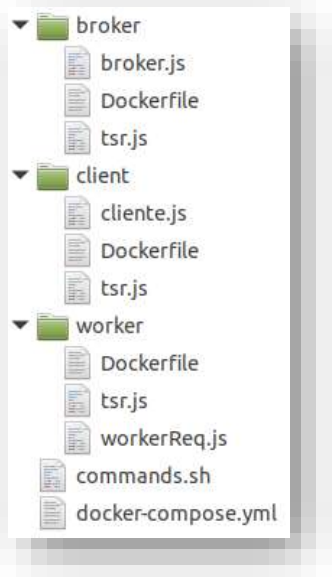
Es un archivo que sigue la sintaxis YAML

Además de las órdenes análogas a los parámetros de "docker run", las principales son:

- **image**: referencia local o remota a una imagen, por nombre o tag
- **build**: ruta a un directorio que contiene un Dockerfile
- **command**: cambia la orden a ejecutar en el inicio
- **links**: enlace a contenedores de otro servicio.
- **external_links**: enlaces a contenedores externos a compose
- **ports**: puertos expuestos (mejor expresarlos entre comillas)
- **expose**: Ídem, pero accesible sólo a servicios enlazados (con links)
- **volumes**: monta rutas como volúmenes

6.7.4 Completando el ejemplo client/broker/worker (aplicación cbw)

En el ejemplo con client, broker y worker desplegado anteriormente, se ha resuelto *artesanalmente* la dependencia del resto de componentes respecto al broker: necesitaban conocer su IP.



Este método es incompatible con la automatización del despliegue, incluso con el momento en que cada información se requiere: ¡necesitamos ***fijar*** el broker antes de poder desplegar el resto!

- Si se cambiara la IP del broker, ¿habría que desplegar de nuevo client y worker?

Debe haber una solución mejor, y ésa es una de las aportaciones de docker-compose y su fichero de descripción del despliegue.

Supongamos que el broker pueda anunciar detalles sobre sí mismo (como su IP), y que existe una forma de inyectar esa información en los componentes adecuados. Esto lo podemos especificar en un archivo de configuración del despliegue docker-compose.yml:

```
version: '2'
services:
  cli:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_HOST=bro
      - BROKER_PORT=9998
  wor:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_HOST=bro
      - BROKER_PORT=9999
```

```
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
```

También necesitamos que los Dockerfile de client y worker puedan consultar esa información. Observaréis que hemos colocado el URL completo para acceder al socket ZMQ del broker, por lo que la última línea del Dockerfile de los componentes debería cambiarse por la siguiente:

- En el Dockerfile del cliente:

```
CMD node myclient $BROKER_HOST $BROKER_PORT
```

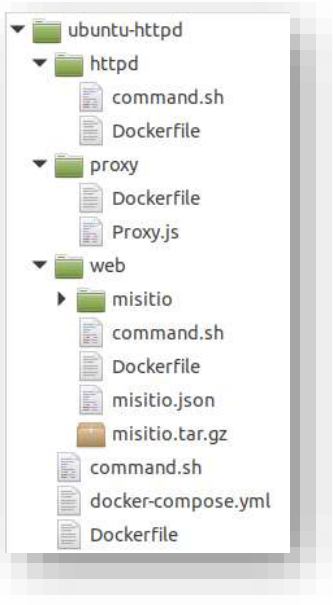
- En el Dockerfile del trabajador:

```
CMD node myworker $BROKER_HOST $BROKER_PORT
```

6.7.5 Uniendo TCPProxy y el servidor de web (aplicación proxyweb)

La introducción de un elemento como el proxy TCP entre otro proceso (el servidor de web) y el exterior (los clientes) debe considerar algunos aspectos de interés:

Interesa que los clientes no puedan percibir ninguna diferencia entre el acceso anterior al servidor y el actual, que atraviesa un proxy. Esto supone que el *endpoint* de servicio debe ser mantenido, y nosotros deberemos preocuparnos de los detalles:



- El URL de invocación será capturado por el proxy
- El punto de entrada del servidor será usado como destino por el proxy
- El punto de entrada del servidor NO podrá ser accedido desde el exterior

Como ya se ha adelantado en el apartado anterior, desplegar un componente que depende de otro obliga a algún ajuste en el archivo de configuración de despliegue del componente (Dockerfile) y del despliegue del servicio (docker-compose.yml). Vamos a nombrar al primer parámetro (puerto del servidor) WEB_PORT, y al segundo (IP del servidor) WEB_IP.

En resumen, el Dockerfile para el Proxy será:

```
FROM tsr-zmq
COPY ./ProxyConf.js Proxy.js
EXPOSE 80
CMD node Proxy $WEB_PORT $WEB_IP
```

En el directorio en que se encuentra tanto el Dockerfile como el archivo ProxyConf.js, ejecutamos:

```
docker build -t proxy .
```

docker-compose.yml

Necesitamos una sección para el componente proxy, destacando:

a) Asociación con el componente que hemos creado (orden image)
b) Acceso a la información creada para el componente web en su despliegue (valor web para la orden links)
c) Registro para usar un puerto (8000) del anfitrión como si se tratara del 80 (variable LOCAL_PORT) del contenedor (valor "8000:80" para la orden ports)

Y también necesitamos "ocultar" el puerto 80 del componente web para impedir que pueda ser accedido directamente sin pasar por el proxy.

a) Retirar orden ports de la sección web
b) Colocar valor "80" para la orden expose en la sección web (permite que sea utilizado internamente)

```
version: '2'
  services:
    proxy:
      image: proxy
      links:
        - web
      ports:
        - "8000:80"
      environment:
        - WEB_PORT=80
        - WEB_IP=web
    web:
      image: misitio
      command: /usr/sbin/apache2ctl -D FOREGROUND
      expose:
        - "80"
```

Si se han seguido los pasos anteriores, y nos colocamos en el directorio que contiene este fichero de configuración, para hacer funcionar la magia y arrancar una instancia de cada uno de los componentes ejecutaremos:

```
docker-compose up
```

6.8 Múltiples nodos

El objetivo de diseño de Compose se limita a componentes que has de ejecutar en un único nodo; sin embargo, la escalabilidad no puede proceder del reparto de los recursos de un nodo entre los componentes de la aplicación, sino de la agregación de otros nodos con sus recursos a nuestra aplicación.

- La tolerancia a fallos, con un solo nodo, queda completamente desnaturalizada.
- La mera concepción de una aplicación distribuida limitada a un nodo es contradictoria.

¿Qué se podría necesitar para que los sistemas Docker de múltiples nodos puedan interactuar e integrarse como si de un sistema único se tratara? Un director de orquesta que los coordine.

- El software de **orquestación** pondrá en contacto a todos los nodos entre sí, ofreciendo propiedades al sistema y funcionalidad a las aplicaciones.

En general, las aplicaciones más veteranas diseñadas con este propósito nacieron en el entorno de la virtualización y de la nube. Algunos casos destacables son Kubernetes (de Google) y Apache Mesos. Con la llegada de las tecnologías de contenerización, estos sistemas también se han adaptado para interactuar con Docker, pero tienen que rivalizar con la propuesta nativa denominada "Docker en modo **Swarm**", incorporada a partir de la versión 1.12[22] de Docker.

- El vencedor es claramente Kubernetes, de tal forma que el equipo de Docker decidió facilitar la integración con este sistema, reduciendo las posibilidades de éxito de Swarm en este ámbito.

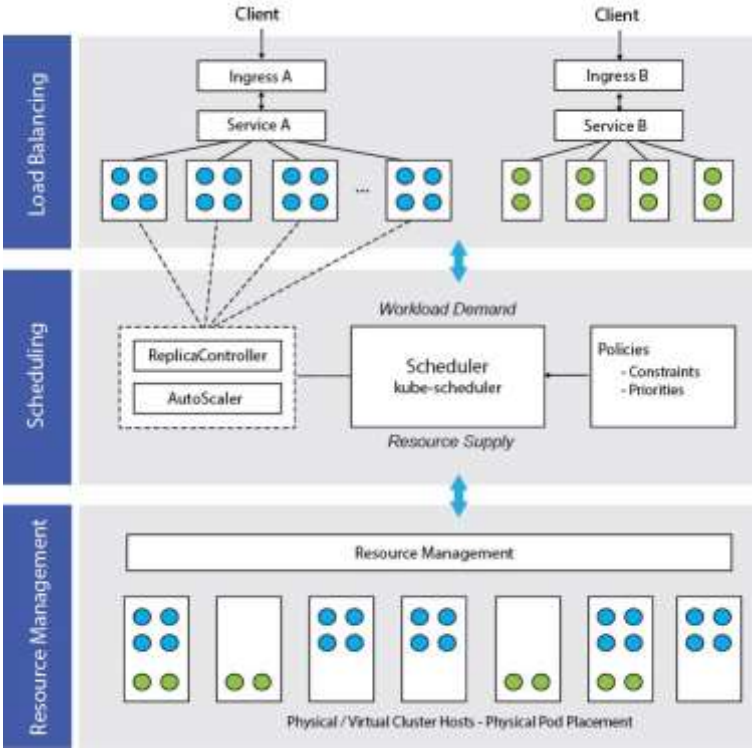
Pese a que nuestro tiempo de laboratorio es limitado y excluye el uso de esta opción multinodo, creemos imprescindible conocer los elementos más relevantes de k8s.

7 KUBERNETES

Kubernetes es un software de código libre que comenzó en 2014 a partir de un proyecto (**Borg**) de Google diseñado para automatizar el despliegue y escalado de aplicaciones contenerizadas. K8s, que es su abreviatura, es el orquestador de contenedores más respaldado en la comunidad, destacando organizaciones y empresas de primer orden como Linux Fundation, Cisco, Docker, Google, Redhat, IBM, Microsoft, Oracle, Suse, Vmware, Ebay, SAP y Yahoo!.

K8s proporciona todo lo necesario para mantener en producción nuestra aplicación distribuida, de forma portable, extensible y automatizable. Se encarga de …

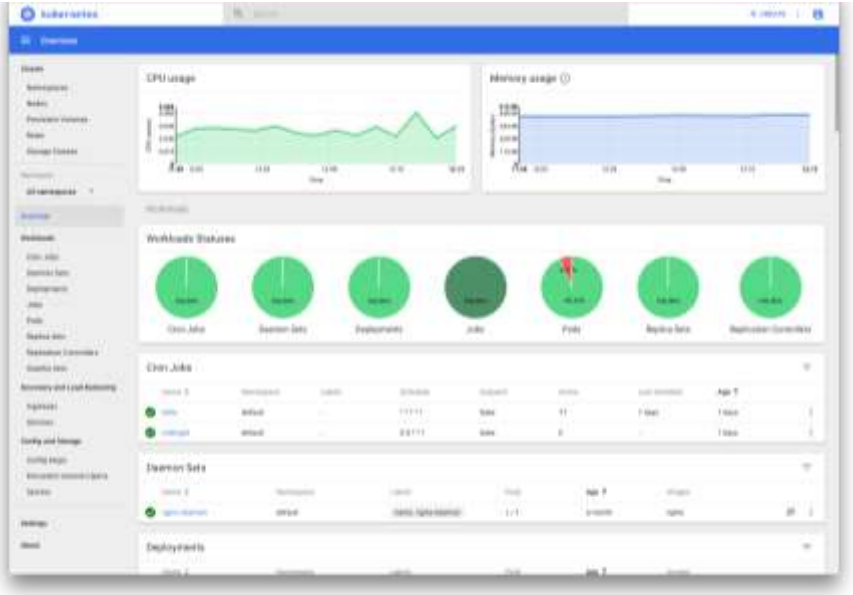
- Montaje de volúmenes para su persistencia (múltiples posibilidades).
- Distribución de secretos y gestor de la configuración.
- Gestión del ciclo de vida de los contenedores.
- Replicación de contenedores.
- Autoscalado horizontal.
- Descubrimiento y equilibrado de servicios.
- Monitorización.
- Acceso a anotaciones de seguimiento y depuración.



[22] Anteriormente existía un Swarm-kit limitado e incompatible con el sistema actual

Existen múltiples herramientas construidas alrededor de k8s, destacando…

- Kubectl (interfaz de línea de órdenes para controlar el cluster)
- Kubeadm (interfaz para aprovisionar un cluster sobre servidores virtuales o reales)
- Kubefed (interfaz para administrar una federación de clusters)
- Minikube (herramienta que facilita la construcción de un cluster k8s mononodo local para desarrollo)
- Dashboard (interfaz web para desplegar y gestionar aplicaciones contenerizadas de un cluster k8s)



- Helm (https://helm.sh/, gestión de paquetes con recursos k8s preconfigurados, llamados charts)
- Kompose (facilita el paso de Docker Compose a k8s)

7.1 Elementos clave de k8s

A modo introductorio, en k8s encontramos:

- **Cluster**: Conjunto de máquinas físicas o virtuales y otros recursos utilizados por k8s.
- **Nodo**: Una máquina física o virtual ejecutándose en k8s en la que se pueden programar *pods*.
- **Pod**: Es la unidad más pequeña desplegable que puede ser creada, programada y manejada por k8s. Es un grupo de uno o más contenedores con almacenamiento compartido entre ellos (misma máquina) y las opciones específicas de cada uno para ejecutarlos.
    - Los contenedores del mismo pod son visibles entre sí mediante *localhost*.
    - En términos de Docker, un pod es un conjunto de contenedores con *namespace* y volúmenes compartidos.
- **Controlador de replicación** (*replication controller*, que abreviamos como *rc*): Se encarga del ciclo de vida de un grupo de pods, y define sus políticas de restauración. Asegura que esté ejecutándose la cantidad especificada de réplicas del pod. Permite escalar de forma fácil los sistemas y maneja la recuperación de un pod cuando ocurre un fallo.
- **Controlador de despliegue** (*deployment controller*): Se encarga de la actualización de una aplicación distribuida.
- **Servicio**: Es una abstracción que define un conjunto de pods y la lógica para acceder a ellos.

- **Espacios de nombres** (*namespaces*): Establece un nivel adicional de separación entre contenedores que comparten recursos de un clúster.
- ***Configmap***: Servicio para gestionar la configuración de las aplicaciones.
- **Secretos**: Servicio para gestionar información privada (p.ej. credenciales) de nuestras aplicaciones.
- **Volúmenes**: Servicio para gestionar la persistencia de los contenedores.

7.2 Trabajando con k8s

En este apartado mencionaremos las operaciones básicas sobre pods, controladores de replicación, servicios y acceso web.

7.2.1 Pods

Los pods son **entidades efímeras** con un ciclo de vida que, en su creación, les asigna un UID válido hasta que los pods terminen o se eliminen. Un pod puede ser reemplazado en otro nodo, recibiendo un nuevo UID.

Los pods pueden utilizarse para el escalado horizontal, pero es preferible disponer de microservicios en contenedores diferentes para conseguir un sistema distribuido más robusto.

Los pods se pueden crear mediante kubectl, directamente por línea de órdenes o a través de un fichero de tipo YAML. Vemos ambas alternativas a continuación.

Creación de un pod por línea de órdenes

Usamos kubectl para interactuar por línea de órdenes con la API de k8s:

```
kubectl run my-nginx --image=nginx --port=80
```

- El primer parámetro indica la acción (arrancar un pod)
- Después el nombre que queremos asignarle (en este caso *my-nginx*)
- El tercero es la imagen a partir de la que se va a instanciar el pod (se llama *nginx*)
- Por último, el puerto en el que escuchará

Nota: la creación de un pod por línea de órdenes incluye implícitamente un rc que se encargue de restaurar el pod cuando sea eliminado, porque la política de restauración por defecto es *Always*

Para consultar información del pod y del rc ejecutamos…

```
kubectl get pods
kubectl get rc
```

```
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get pods
NAME             READY     REASON    RESTARTS   AGE
my-nginx-48onk   1/1       Running   0          2m
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR       REPLICAS
my-nginx     my-nginx       nginx      run=my-nginx   1
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
```

Puede comprobarse que el nombre del pod coincide con el del rc mencionado en la orden (*my-nginx*) seguido de un identificador único para cada pod.

Para eliminar el controlador de replicación junto con el pod ejecutamos…

```
kubectl delete rc my-nginx
```

Creación de un pod mediante un fichero YAML

- **/opt/kubernetes/examples/nginx/nginx-pod.yaml**

```
# Número de versión de la API a utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: Pod
# Datos propios del pod como el nombre y las etiquetas que tiene asociados para seleccionarlo
metadata:
    name: my-nginx
    # Especificamos que el pod tenga una etiqueta con clave "app" y valor "nginx"
    labels:
        app: nginx
# Contiene la especificación del pod
spec:
    # Aquí se nombran los contenedores que forman parte de este pod, todos visibles por
localhost
    containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
    # Política de recuperación si el pod se detiene o termina por a un fallo interno.
    restartPolicy: Always
```

Procedemos a crear el mismo pod pero desde el fichero que acabamos de crear:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-pod.yaml
```

En esta modalidad podemos observar que no se crea ningún controlador de replicación.

```
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pod
NAME       READY     REASON    RESTARTS   AGE
my-nginx   1/1       Running   0          1m
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

Eliminamos este pod mediante…

```
kubectl delete pod my-nginx
```

7.2.2 Controlador de replicación

El **controlador de replicación** (rc = *replication controller*) se usa para asegurar que siempre haya algún pod disponible. Si hay muchos pods, eliminará algunos; si hay pocos, creará nuevos.

Creación de un rc

En este ejemplo creamos un rc llamado **nginx-rc.yaml** encargado de ejecutar un servidor nginx:

- **/opt/kubernetes/examples/nginx/nginx-rc.yaml**

```
# Número de versión de la API que se quiere utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: ReplicationController
# Datos propios del controlador de replicación
metadata:
```

```
    # Nombre del controlador de replicación
    name: my-nginx
# La especificación del estado deseado que queremos que tenga el pod.
spec:
    # Número de réplicas que queremos que el rc mantenga (esto creará un pod)
    replicas: 1
    # En esta propiedad se indican todos los pods que este rc gestionará.En este caso, todos los
que tengan el valor "nginx" en la etiqueta "app"
    selector:
        app: nginx
    # Esta propipedad tiene el mismo esquema interno que un pod, pero no necesita "apiVersion"
ni "kind" porque ya ha sido especificado
    template:
        metadata:
            name: nginx
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx
                  ports:
                    - containerPort: 80
```

Ahora creamos el rc con la orden:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
```

Vemos cómo ahora se nos ha creado un pod con un UID.

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pods
NAME             READY     REASON    RESTARTS   AGE
my-nginx-l15ux   1/1       Running   0          1m
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

También podemos comprobar el estado de nuestro controlador de replicación (nombre, número de pods y sus respectivos estados, ...) a través de la orden kubectl:

```
kubectl describe rc my-nginx
```

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl describe rc my-nginx
W0318 10:00:38.855526   10502 request.go:302] field selector: v1 - events - involvedObject.namespace - default: need to check if this is versioned correctly.
W0318 10:00:38.856097   10502 request.go:302] field selector: v1 - events - involvedObject.kind - ReplicationController: need to check if this is versioned correctly.
W0318 10:00:38.856432   10502 request.go:302] field selector: v1 - events - involvedObject.uid - d18350fd-ecef-11e5-a835-0800271a0b5b: need to check if this is versio
W0318 10:00:38.856710   10502 request.go:302] field selector: v1 - events - involvedObject.name - my-nginx: need to check if this is versioned correctly.
Name:           my-nginx
Image(s):       nginx
Selector:       app=nginx
Labels:         app=nginx
Replicas:       1 current / 1 desired
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen                     LastSeen                      Count   From                        SubobjectPath   Reason                  Message
  Fri, 18 Mar 2016 09:57:26 +0000       Fri, 18 Mar 2016 09:57:26 +0000 1       {replication-controller }                   successfulCreate        Created pod: m
```

Trabajando con controladores de replicación

Una vez creado, un rc te permite:

- **Escalarlo:** puedes escoger el número de réplicas que tiene un pod de forma dinámica.
- **Eliminar el rc:** puedes borrar solo el controlador de replicación o borrarlo junto a todos los pods de los que se encarga
- **Aislar al pod del rc:** Los pods pueden no pertenecer a un controlador de replicación cambiando las etiquetas. El pod eliminado será sustituido por otro nuevo que creará el rc.

Para añadir un pod controlado por el rc ejecutamos…

```
kubectl scale rc my-nginx --replicas=2
```

Para eliminar un rc ejecutamos…

```
kubectl delete rc my-nginx
```

7.2.3 Servicios

Como ya sabemos, los pods son elementos volátiles con una dirección IP que puede cambiar. Los servicios permiten que un pod pueda comunicarse con otro.

Un servicio es una **abstracción que define un grupo lógico de pods y una política de acceso a los mismos**. Los pods apuntan a un servicio mediante la propiedad label. En el caso anterior[23], si algo causara la destrucción de este pod, el rc crearía uno nuevo con una nueva IP, de forma que el resto de la infraestructura que dependiera de ese pod por esa IP fija dejaría de funcionar. **El servicio consigue que ese pod siempre sea accesible de la misma manera**. Para ilustrarlo, crearemos un servicio y lo haremos accesible fuera del cluster.

Creación de un servicio

- **/opt/kubernetes/examples/nginx/nginx-svc.yaml**

```
# Número de versión de la API a utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: Service
# Datos propios del pod como el nombre y las etiquetas asociados para seleccionarlo
metadata:
    name: my-nginx-service
# Contiene la especificación del pod
spec:
    # En esta propiedad se indican todos los pods que apuntan a este servicio. En este caso, se va a encargar de todos los que tengan el valor "nginx" en la etiqueta "app"
    selector:
        app: nginx
    ports:
      # Indica el puerto en el que este servicio atiende peticiones
      - port: 80
```

Una vez creado el fichero levantamos el servicio con la orden:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-svc.yaml
```

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl create -f /opt/kubernetes/examples/nginx/nginx-svc.yaml
services/my-nginx-service
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pods
NAME      READY     REASON    RESTARTS   AGE
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get svc
NAME               LABELS                                    SELECTOR    IP(S)           PORT(S)
kubernetes         component=apiserver,provider=kubernetes   <none>      192.168.3.1     443/TCP
my-nginx-service   <none>                                    app=nginx   192.168.3.110   80/TCP
```

Ahora habilitamos el rc de este servicio, que es el que hemos creado anteriormente:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
```

[23] con un `rc` encargado de ejecutar un pod con un contenedor `nginx`

Para acceder a la aplicación del pod desde dentro del cluster necesitamos saber la IP que tiene actualmente el pod. Usamos la orden kubectl:

```
kubectl get -o template pod my-nginx-m5mni --template={{.status.podIP}}
```

A continuación accedemos al servicio mediante esa IP usando un cliente HTTP (p.ej. el programa curl).

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get -o template pod my-nginx-m5mni --template={{.sta
172.17.0.1root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# curl 172.17.0.1:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

De momento este pod solo es accesible dentro del cluster. El servicio nos sirve para abstraernos del pod en cuestión que estamos utilizando, permitiendo disponer de múltiples aplicaciones de *backend* detrás de un único *frontend*.

Acceso a un servicio desde fuera del cluster

Necesitaremos añadir la propiedad **type: NodePort** al servicio, exponiendo el servicio en cada nodo del cluster, y consiguiendo contactar con el servicio desde cualquier IP de los nodos. Nuestro servicio quedaría de la siguiente manera:

- **/opt/kubernetes/examples/nginx/nginx-svc.yaml**

```
# Número de versión de la API a utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: Service
# Aquí van los datos propios del pod como el nombre y sus etiquetas
metadata:
    name: my-nginx-service
# Contiene la especificación del pod
spec:
    type: NodePort
    # En esta propiedad se indican todos los pods que apuntan a este servkice. En este caso, se
va a encargar de todos los que tengan el valor "nginx" en el label "app"
    selector:
        app: nginx
    ports:
```

```
    # Indica el puerto en el que se debería de servir este servicio
    - port: 80
```

Eliminamos el anterior servicio y añadimos el nuevo con las órdenes que ya sabemos, y comprobamos qué puerto externo se ha asociado con la conexión:

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl delete svc my-nginx-service
services/my-nginx-service
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# vim nginx-svc.yaml
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl create -f nginx-svc.yaml

                            You have exposed your service on an external port on all nodes in your cluster.
                            If you want to expose this service to the external internet, you may need to set up
                            firewall rules for the service port(s) (tcp:80) to serve traffic.

                            See https://github.com/GoogleCloudPlatform/kubernetes/tree/master/docs/services-firewall.md for
                            services/my-nginx-service
```

```
Name:                   my-nginx-service
Labels:                 <none>
Selector:               app=nginx
Type:                   NodePort
IP:                     192.168.3.158
Port:                   <unnamed>       80/TCP
NodePort:               <unnamed>       32138/TCP
Endpoints:              172.17.0.1:80
Session Affinity:       None
No events.
```

Vemos que la conexión se asocia al puerto 32138 (automático). Ahora tendríamos que cambiar la configuración del anfitrión para permitir el acceso a este puerto desde localhost u otra IP (los detalles dependen de cada caso).

Por último accedemos desde el navegador a http://localhost:32138 (puede tardar un poco en mostrarse)

7.2.4 Acceso a la interfaz gráfica

K8s cuenta con una serie de añadidos (*add-ons*) accesibles a través de su API. Nos interesa usar uno, la interfaz gráfica, para lo que en primer lugar habilitamos el *namespace* kube-system, (en el fichero **/opt/kubernetes/cluster/ubuntu/namespace.yaml**). Ejecutamos…

```
kubectl create -f /opt/kubernetes/cluster/ubuntu/namespace.yaml
```

Ahora habilitamos la interfaz gráfica, creando el rc y el servicio que vienen en el directorio **/opt/kubernetes/cluster/addons/dashboard**, pero previamente hemos de exponer el servicio

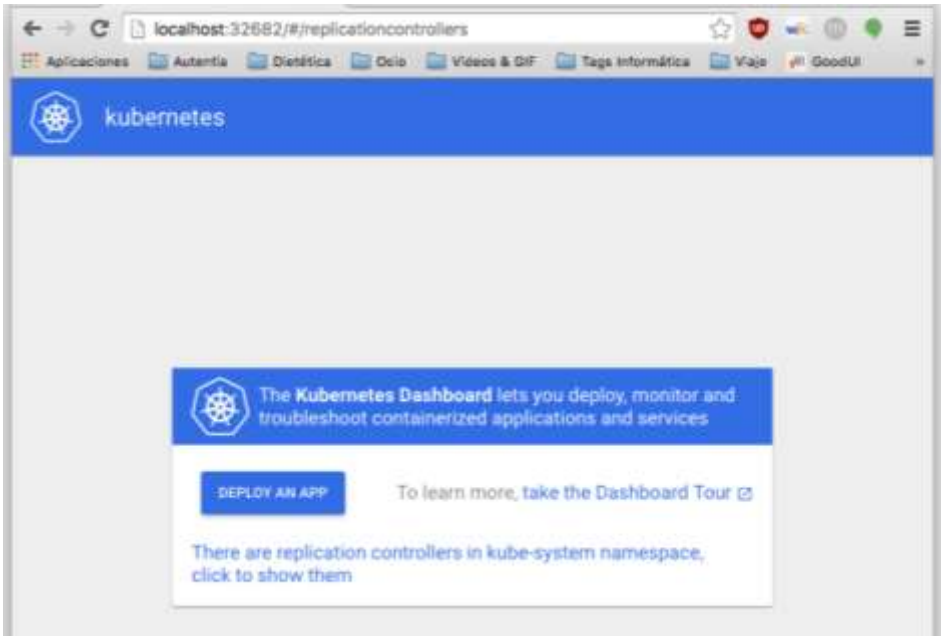
fuera del cluster para poder acceder desde el navegador. Para ello simplemente añadimos **type: NodePort** justo después del spec.

- **/opt/kubernetes/cluster/addons/dashboard/dashboard-service.yaml**

```
apiVersion: v1
kind: service
metadata:
  name: kubernetes-dashboard
  namespace: kube-system
  labels:
    k8s-app: kubernetes-dashboard
    kubernetes.io/cluster-servicio: "true"
spec:
  type: NodePort
  selector:
    k8s-app: kubernetes-dashboard
  ports:
  - port: 80
    targetPort: 9090
```

Ahora ya estamos listos para iniciar la interfaz gráfica con las siguientes órdenes…

```
kubectl create -f dashboard-controller.yaml
kubectl create -f dashboard-service.yaml
```



7.3 Referencias para Kubernetes

- **Tutorial Online oficial:** En https://kubernetes.io/docs/tutorials/ podéis realizar un curso interactivo para correr un minikube en vuestro equipo y aplicaciones para vuestras pruebas y aprender los conceptos básicos.
  - Especialmente la **parte básica** (https://kubernetes.io/docs/tutorials/kubernetes-basics/) compuesta por 5 pasos:
    - Crear un cluster k8s (/cluster-intro/)
    - Desplegar una aplicación (/deploy-intro/)
    - Explorar la aplicación (/explore-intro/)

    - Exponer la aplicación (/expose-intro/)
    - Escalar la aplicación (/scale-intro/)
    - Actualizar la aplicación (/update-intro/)
- Curso Katakoda: https://www.katacoda.com/courses/kubernetes

También pueden encontrarse varias opciones para experimentar con k8s:

- **Minikube** es el método recomendado para crear un clúster de k8s local de un solo nodo para desarrollo y pruebas. La configuración es completamente automática y no requiere una cuenta de proveedor de la nube.
- **Kubeadm-dind** es un clúster k8s multinodo que solo requiere un proceso docker.
- **PWK** El sitio PWK (http://play-with-k8s.com/) permite montar clústers de k8s y lanzar servicios replicados de manera rápida y sencilla durante un máximo de 4 horas. Se trata de un entorno donde disponemos de varias instancias de Docker sobre las que podemos usar kubeadm para instalar y configurar k8s, creando un clúster en menos de un minuto.

8 APÉNDICES

8.1 Código de client/broker/worker

Con parámetros desde la línea de órdenes (y valores por omisión) en las condiciones comentadas anteriormente en este documento.

myclient.js

```
01: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
02: lineaOrdenes("brokerHost brokerPort")
03: let req = zmq.socket('req')
04: let id  = "C_"+require('os').hostname()
05: req.identity = id
06: conecta(req, brokerHost, brokerPort)
07: req.send("C_"+require('os').hostname())
08: function procesaRespuesta(msg) {
09:         traza('procesaRespuesta','msg',[msg])
10:         adios([req], `Recibido: ${msg}. Adios`)()
11: }
12: req.on('message', procesaRespuesta)
13: req.on('error', (msg) => {error(`${msg}`)})
14: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

mybroker.js

```
01: const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
02: const ans_interval = 2000 // deadline to detect worker failure
03: lineaOrdenes("frontendPort backendPort")
04: let failed   = {}  // Map(worker:bool) failed workers has an entry
05: let working  = {}  // Map(worker:timeout) timeouts for workers executing tasks
06: let ready    = []  // List(worker) ready workers (for load-balance)
07: let pending  = []  // List([client,message]) requests waiting for workers
08: let frontend = zmq.socket('router')
09: let backend  = zmq.socket('router')
10: function dispatch(client, message) {
11:   traza('dispatch','client message',[client,message])
12:   if (ready.length) new_task(ready.shift(), client, message)
13:   else pending.push([client,message])
14: }
15: function new_task(worker, client, message) {
16:   traza('new_task','client message',[client,message])
```

```
17:   working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
18:   backend.send([worker,'', client,'', message])
19: }
20: function failure(worker, client, message) {
21:   traza('failure','client message',[client,message])
22:   failed[worker] = true
23:   dispatch(client, message)
24: }
25: function frontend_message(client, sep, message) {
26:   traza('frontend_message','client sep message',[client,sep,message])
27:   dispatch(client, message)
28: }
29: function backend_message(worker, sep1, client, sep2, message) {
30:   traza('backend_message','worker sep1 client sep2
    message',[worker,sep1,client,sep2,message])
31:   if (failed[worker]) return  // ignore messages from failed nodes
32:   if (worker in working) { // task response in-time
33:     clearTimeout(working[worker]) // cancel timeout
34:     delete(working[worker])
35:   }
36:   if (pending.length) new_task(worker, ...pending.shift())
37:   else ready.push(worker)
38:   if (client) frontend.send([client,'',message])
39: }
40: frontend.on('message', frontend_message)
41:  backend.on('message', backend_message)
42: frontend.on('error'   , (msg) => {error(`${msg}`)})
43:  backend.on('error'   , (msg) => {error(`${msg}`)})
44:  process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
45:
46: creaPuntoConexion(frontend, frontendPort)
47: creaPuntoConexion( backend,  backendPort)
```

myworker.js

```
01: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
02: lineaOrdenes("brokerHost brokerPort")
03: let req = zmq.socket('req')
04: let id  = "W_"+require('os').hostname()
05: req.identity = idpasos
06: conecta(req, brokerHost, brokerPort)
07: req.send(['','',''])
08: function procesaPeticion(cliente, separador, mensaje) {
09:         traza('procesaPeticion','cliente separador mensaje',[cliente, separador, mensaje])
10:         setTimeout(()=>{req.send([cliente,'',`${mensaje} ${id}`])}, 1000)
11: }
12: req.on('message', procesaPeticion)
13: req.on('error', (msg) => {error(`${msg}`)})
14: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

8.2 Pequeños trucos para Docker

1. **Identificadores más recientes**

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit $(dl) helloworld
```

2. **Consultar IP**

```
docker inspect $(dl) | grep IPAddress | cut -d '"' -f 4
```

3. **Asignar puerto**

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf
0).HostPort}} {{end}}' <id_contenedor>
```

4. **Localizar contenedores mediante expresión regular**

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done
```

5. **Consultar entorno**

```
docker run --rm ubuntu env
```

6. **Finalizar todos los contenedores en ejecución**

```
docker kill $(docker ps -q)
```

7. **Eliminar contenedores viejos**

```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

8. **Eliminar contenedores detenidos**

```
docker rm -v $(docker ps -a -q -f status=exited)
```

9. **Eliminar imágenes *colgadas***

```
docker rmi $(docker images -q -f dangling=true)
```

10. **Eliminar todas las imágenes**

```
docker rmi $(docker images -q)
```

11. **Eliminar volúmenes *colgadas***

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

(En 1.9.0, dangling=false no funciona y muestra todos los volúmenes)

12. **Mostrar dependencias entre imágenes**

```
docker images -viz | dot -Tpng -o docker.png
```

13. **Monitorizar el consumo de recursos al ejecutar contenedores**

Para averiguar el consumo de CPU, memoria y red de un único contenedor, puedes usar:

```
docker stats <container>
```

Para todos los contenedores, ordenados por id:

```
docker stats $(docker ps -q)
```

Ídem ordenados por nombre:

```
docker stats $(docker ps --format '{{.Names}}')
```

Ídem seleccionando los que proceden de una imagen:

```
docker ps -a -f ancestor=ubuntu
```

8.3 Docker-compose.yml

Este anexo **no** es exhaustivo, y docker-compose se encuentra activamente **en evolución**, de manera que la documentación original es una fuente irrenunciable de información y referencia.

- Dispones de material de referencia en la "**Docker reference documentation**" de este mismo tema.

El archivo docker-compose.yml especifica las características para el despliegue mediante docker-compose de una aplicación distribuida. Su contenido son líneas de texto que se ajustan a una sintaxis especificada a partir de YAML (https://docs.docker.com/compose/yml/). A destacar:

- No pueden usarse tabuladores, tan solo espacios en blanco.
- Las propiedades y listas se deben indentar con un espacio o más.
- Las mayúsculas y minúsculas son significativas tanto en los identificadores de propiedades como en los de claves.

Cada uno de los servicios que se definan en docker-compose.yml debe especificar exactamente una imagen. El resto de claves son opcionales, y son análogas a sus correspondientes para la orden run de Docker. Las órdenes incluidas en el Dockerfile no necesitan repetirse en docker-compose.yml.

image

Identificador de la imagen, local o remota. Composer intentará obtenerla (*pull*) si no se dispone de ella localmente.

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

build

Ruta a un directorio con el Dockerfile. Si se trata de una ruta relativa, lo será respecto a la del archivo yml. Compose lo construirá y designará con un nombre.

```
build: ./dir

build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

dockerfile: Ruta del archivo dockerfile para construir la imagen.

command

Sustituye a la orden por defecto.

```
command: bundle exec thin -p 3000
```

links

Enlaza con contenedores de otro servicio. Especifica el nombre del servicio y el alias para el link (SERVICIO:ALIAS), o solo el nombre si se va a mantener el mismo en el alias.

```
web:
```

```
  links:
   - db
   - db:database
   - redis
```

El nombre del servicio actúa como nombre DNS del servidor salvo que se haya definido un alias.

Los enlaces expresan dependencias entre servicios, e influyen en el orden de arranque del servicio.

external_links

Enlaces a contenedores externos a este compose.yml, o incluso externos a Compose, especialmente para los que ofrecen servicios compartidos. La semántica de esta orden es similar a links cuando se usa para especificar el nombre del contenedor y el alias del enlace (CONTENEDOR:ALIAS).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

ports

Puertos expuestos, bien indicando ambos (EQUIPO:CONTENEDOR), o bien indicando únicamente el del equipo contenedor (se tomará un puerto del anfitrión al azar).

Nota: Se recomienda especificar los puertos y su correspondencia como cadenas con comillas.

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

expose

Expone puertos sin publicarlos al equipo anfitrión, de manera que únicamente tendrán acceso los servicios enlazados dentro de Docker. Solo se puede indicar el puerto interno.

```
expose:
 - "3000"
 - "8000"
```

volumes

Monta rutas como volúmenes, especificando opcionalmente una ruta en el anfitrión (HOST:CONTENEDOR), o un modo de acceso (HOST:CONTENEDOR:ro).

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro
```

```
  # Named volume
  - datavolume:/var/lib/mysql
```

Se puede montar una ruta relativa al anfitrión, que será expandida de forma relativa al directorio del archivo de configuración de Compose que se esté empleando. Las rutas relativas comienzan siempre por . o ..

environment

Añade variables de entorno

Los valores de las variables de entorno que solo disponen de una clave se calculan en la máquina en la que se ejecuta Compose, siendo especialmente útil para valores secretos o a medida del anfitrión.

```
environment:
  RACK_ENV: development
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

labels

Añade metadatos a contenedores mediante labels de Docker. Puede elegirse entre array y diccionario.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

8.4 Otros ejemplos de casos de uso de Docker

Un lugar de referencia inevitable para encontrar ejemplos y aplicaciones contenerizadas es https://hub.docker.com/explore/

En este apartado damos cabida a algunos ejemplos que pretenden ilustrar desde el uso de Docker para recubrir el despliegue de una aplicación gráfica de escritorio (OpenOffice), y un contenedor que actúa como GUI de Docker (portainer).

8.4.1 LibreOffice en un contenedor

Contenido del Dockerfile

```
FROM debian:stretch
MAINTAINER Jessie Frazelle <jess@linux.com>

RUN apt-get update && apt-get install -y \
        libreoffice \
        --no-install-recommends \
        && rm -rf /var/lib/apt/lists/*

ENTRYPOINT [ "libreoffice" ]
```

Invocación en línea de órdenes

```
docker run -d \
  -v /etc/localtime:/etc/localtime:ro \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e DISPLAY=unix$DISPLAY \
  -v $HOME/slides:/root/slides \
  -e GDK_SCALE \
  -e GDK_DPI_SCALE \
  --name libreoffice \
  jess/libreoffice
```

Otra **alternativa**, sin necesidad de Dockerfile (de http://linoxide.com/how-tos/20-docker-containers-desktop-user/)

```
xhost +local:docker
docker run \
-v $HOME/Documents:/home/libreoffice/Documents:rw \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e uid=$(id -u) -e gid=$(id -g) \
-e DISPLAY=unix$DISPLAY --name libreoffice \
chrisdaish/libreoffice
```

8.4.2 Portainer

Ejemplo de aplicación que accede al socket de Docker, mediante el cual puede conocer el estado de los demás contenedores alojados en este anfitrión.

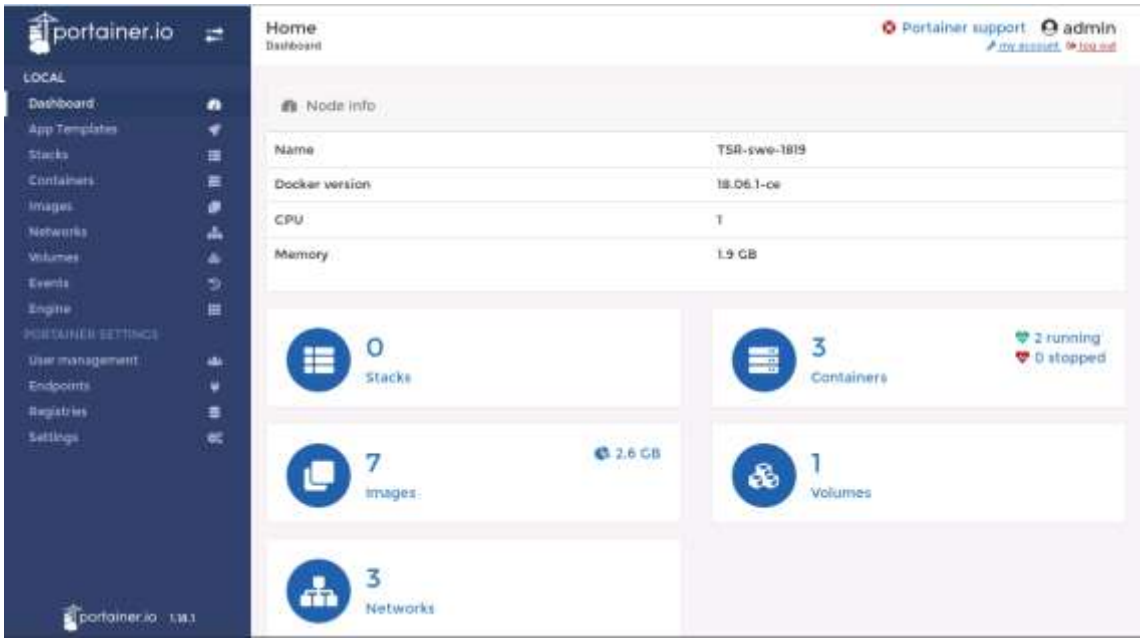
```
#!/bin/bash
# URL: https://portainer.io

#docker volume create portainer_data
docker run -d -p 9999:9000 --name portainer --restart always -v
/var/run/docker.sock:/var/run/docker.sock -v /root/Downloads/.portainer_data:/data
portainer/portainer
```

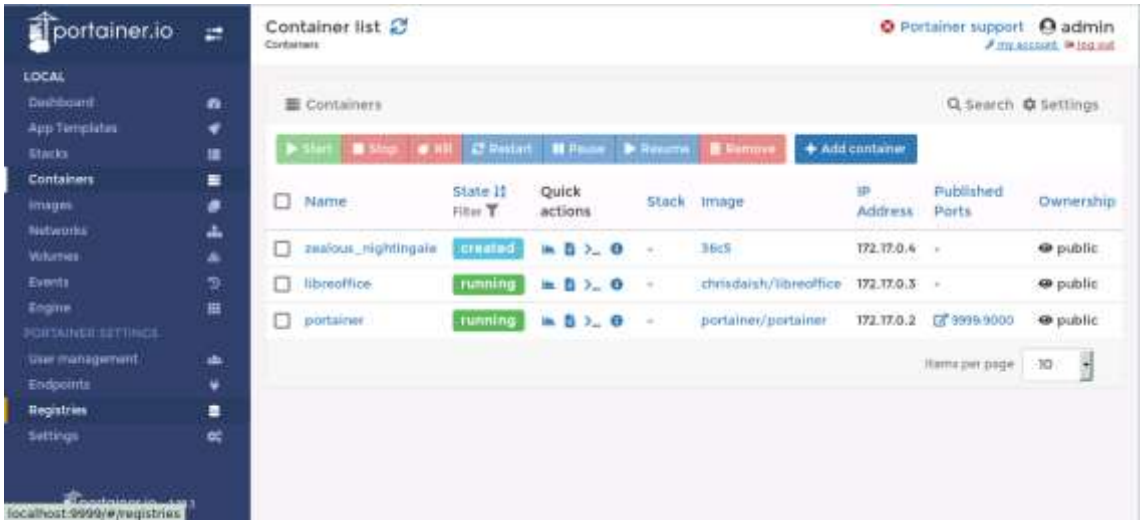
Accedemos con el navegador al URL `http://localhost:9999/`

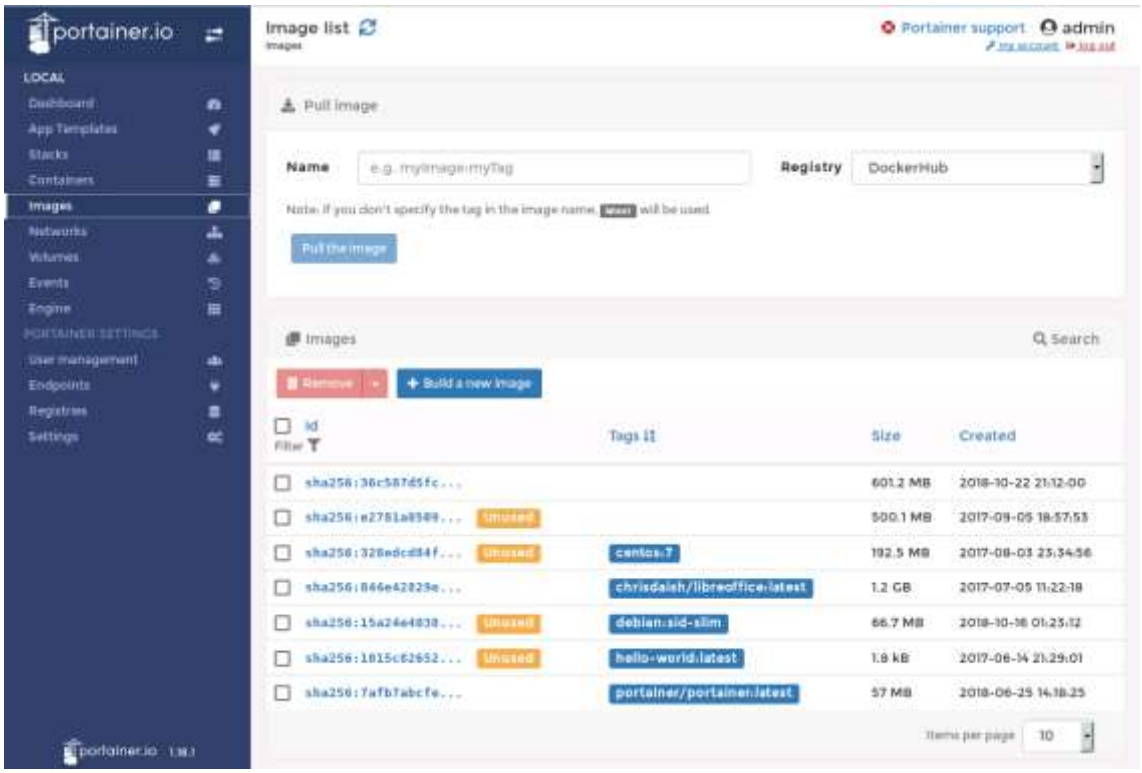
El primer acceso permite crear una contraseña para el usuario admin.

Queda a nuestra disposición el tablero de control (*dashboard*) de la aplicación. Puede observarse un menú a la izquierda para seleccionar apartados dentro de la aplicación.



A título ilustrativo se muestra el aspecto de los apartados *Containers* e *Images* de un caso real:





- Más información en `portainer.io`

9 REFERENCIAS

9.1 En la web

- **www.docker.com** (lugar oficial de Docker) Atención a las diferencias entre las versiones 1, 2 y 3 de Compose, y a la posible confusión entre SwarmKit (obsoleto) y modo Swarm
  - docs.docker.com/userguide/ (**documentación oficial**)
  - docs.docker.com/compose/ (Compose)
    - docker-compose.yml: https://docs.docker.com/compose/compose-file/
  - docs.docker.com/samples/
    - Ejemplos ilustrativos y variados
  - docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
    - Consejos sobre Dockerfile
- **github.com/wsargent/docker-cheat-sheet**
  - Un resumen muy acertado sobre órdenes y ficheros de Docker (sin Compose)
- **github.com/veggiemonk/awesome-docker**
  - Lista exhaustiva de enlaces sobre Docker
- comp.photo777.org/wp-content/uploads/2014/09/Docker-ecosystem-8.6.1.pdf
  - Póster sobre el ecosistema Docker
- www.mindmeister.com/389671722/docker-ecosystem
  - Mapa conceptual sobre Docker/Open Container
- flux7.com/blogs/docker/ (blog sobre Docker)
- 12factor.net/ (metodología de diseño de aplicaciones "*The twelve-factor app*")
- **kubernetes.io**

9.2 Bibliografía

[1] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies," Technical Report ADA452086, Defense Technical Information Center , Dept. of Defense, USA, 1998.

[2] Microsoft Developer Network, "Windows Installer Guide," Microsoft Corp., [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa372845%28v=vs.85%29.aspx. [Accessed 9 March 2016].

[3] Microsoft Developer Network, "Windows Installer Reference," Microsoft Corp., [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa372860%28v=vs.85%29.aspx. [Accessed 9 March 2016].

[4] FireGiant, "WiX Tutorial," Outercurve Foundation, [Online]. Available: https://www.firegiant.com/wix/tutorial/. [Accessed 9 March 2016].

[5] E. Foster-Johnson, "RPM Guide," 2005. [Online]. Available: http://rpm5.org/docs/rpm-guide.html. [Accessed 9 March 2016].

[6] "DNF, the next-generation replacement for Yum," Red Hat, Inc., 2012. [Online]. Available: http://dnf.readthedocs.org/en/latest/. [Accessed 9 March 2016].

[7] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," Research report RC22456, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA, 2002.

[8] W. Li, "Evaluating the impacts of dynamic reconfiguration on the QoS of running systems," *Journal of Systems and Software,* vol. 84, pp. 2123-2138, 2011.

[9] S. Dustdar, Y. Guo, B. Satzger and H. L. Truong, "Principles of Elastic Processes," *IEEE Internet Computing,* vol. 15, no. 5, pp. 66-71, 2011.

[10] R. A. Gingell, M. Lee, X. T. Dang and M. S. Weeks, "Shared Libraries in SunOS," in *USENIX Association Conference*, Atlanta, Georgia, USA, 1987.

[11] D. Nene, "A beginner's guide to Dependency Injection," TheServerSide.COM, 1 July 2005. [Online]. Available: http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection. [Accessed 10 March 2016].