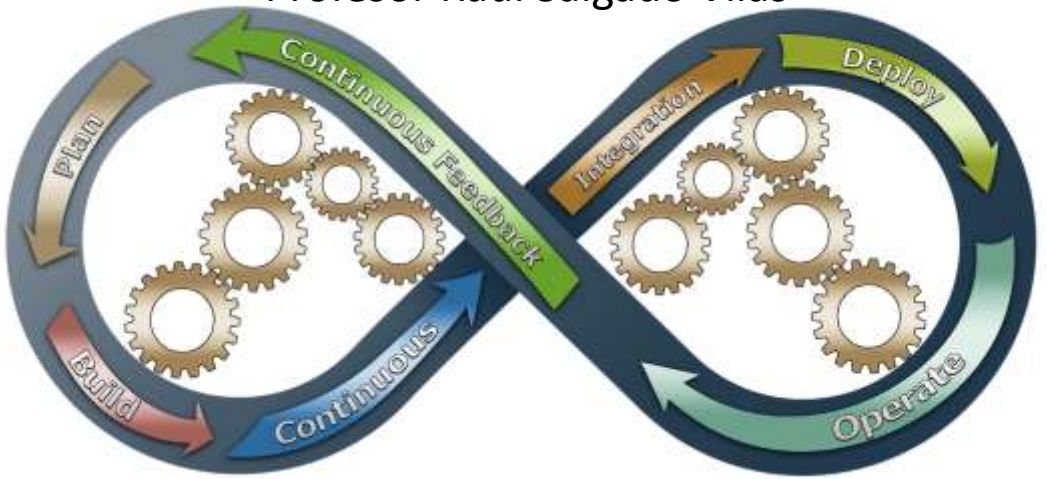
unir LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA

UF-3 CLOUDS, VIRTUALIZACIÓN Y DOCKERS

Profesor Raúl Salgado Vilas

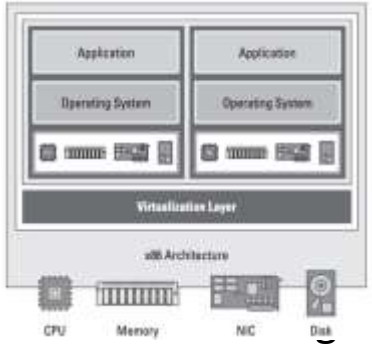


- Virtualización: Un paso más allá a la hora de utilizar los recursos físicos es el de la virtualización: aprovechar los recursos disponibles en un ordenador, o en una red de ordenadores, para generar ‘máquinas virtuales’ que los aíslen y generen un entorno de ejecución o procesamiento.

- La virtualización tiene varios usos comunes y lo cierto es que todos ellos giran en torno al concepto de que su tecnología representa una abstracción de los recursos físicos.

- Hay dos tipos de virtualización basada en hipervisores:

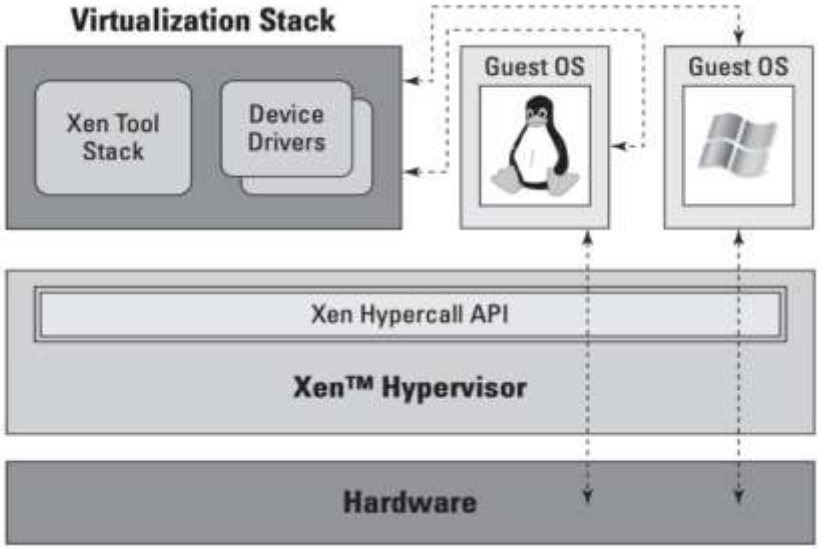
  - ✓ Imitación o emulación del hardware: En este caso, el software de virtualización (hipervisor) crea una máquina virtual que imita todo el entorno de hardware. El sistema operativo que está cargado en una máquina virtual es un producto estándar no modificado. Cuando realiza llamadas para recursos del sistema, el software de emulación de hardware captura la llamada del sistema y la redirige para que pueda gestionar estructuras de datos proporcionadas por el hipervisor. Es el propio hipervisor el que realiza las llamadas al hardware físico real, subyacente a toda la aglomeración de software:



- Hay dos tipos de virtualización basada en hipervisores:

  - ✓ La emulación o imitación de hardware también es conocida como virtualización de metal desnudo (del inglés, bare metal virtualization), para simbolizar el hecho de que ningún software se encuentra entre el hipervisor y el 'metal' del servidor. Como hemos mencionado, el hipervisor intercepta las llamadas del sistema desde la máquina virtual huésped y coordina el acceso al hardware subyacente directamente.

  - ✓ Paravirtualización La paravirtualización no intenta emular un entorno de hardware en software, sino que un hipervisor de paravirtualización coordina (o multiplexa) el acceso a los recursos de hardware subyacentes del servidor. La arquitectura de la paravirtualización son los entornos Xen:



- Hay dos tipos de virtualización basada en hipervisores:

  - ✓ En la paravirtualización, el hipervisor reside en el hardware y, por lo tanto, esta se puede concebir como una arquitectura de virtualización de metal desnudo. Uno o más sistemas operativos huésped (equivalente a máquinas virtuales en virtualización de emulación del hardware) se ejecutan sobre el hipervisor. Un huésped privilegiado se ejecuta como una máquina virtual huésped, pero tiene privilegios que le permiten acceder directamente a ciertos recursos en el hardware subyacente.

- Proveedores de virtualización: VMware, Citrix y Microsoft son los principales proveedores de virtualización de servidores x86 en entornos profesionales, más adelante, veremos otras opciones susceptibles de ser usadas también en entornos personales y de prueba.

  - ✓ VMware Es el proveedor de virtualización de servidores más extendido y afianzado en el mercado. La plataforma insignia de VMware, vSphere, utiliza la tecnología de emulación de hardware.

  - ✓ Citrix Ofrece un producto de virtualización de servidor llamado XenServer basado en paravirtualización. El huésped privilegiado (llamado control domain en lenguaje Xen) y el hipervisor Xen trabajan en equipo para permitir que las máquinas virtuales huésped interactúen con el hardware subyacente.

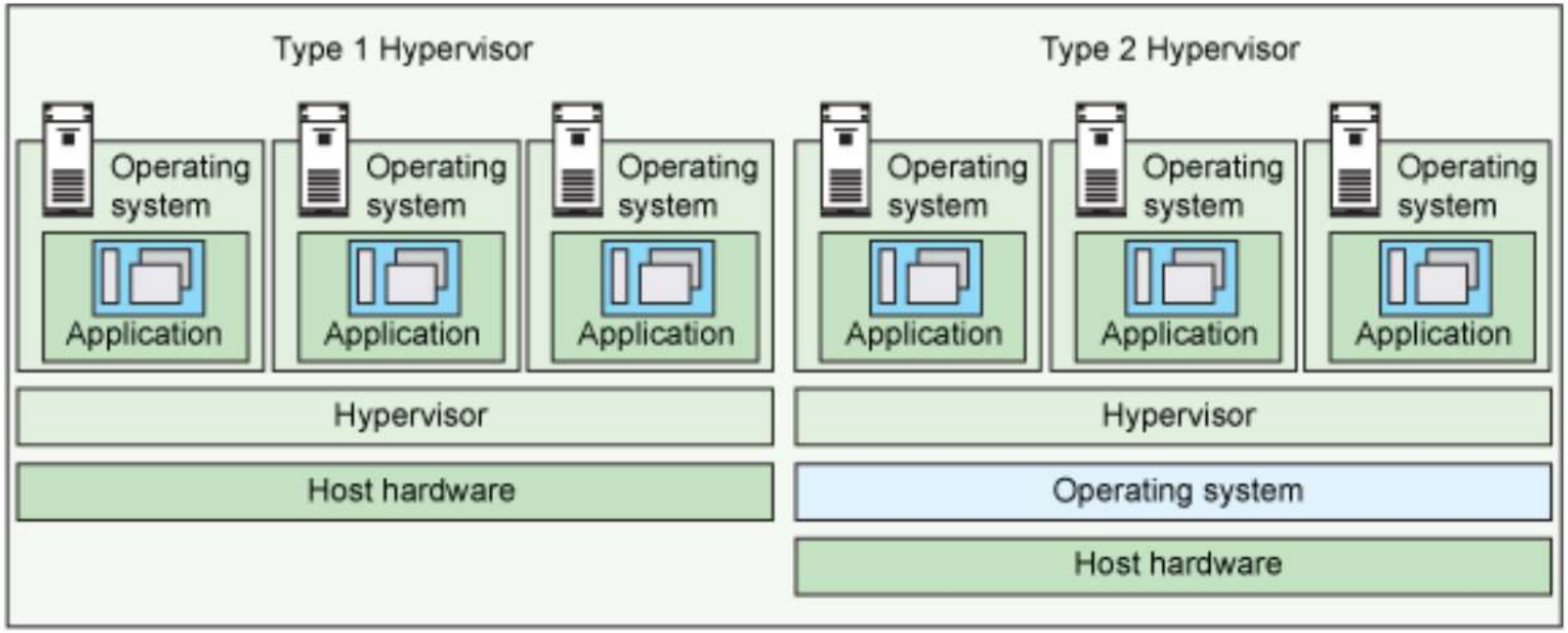
- ✓ Microsoft Hyper-V, el producto de virtualización del servidor de Microsoft tiene una arquitectura muy similar a la de Xen. En lugar de usar el término control domain para referirse a las máquinas virtuales huésped, Hyper-V se refiere a ellas como particiones y a la contraparte del control domain de Xen se la denomina partición principal

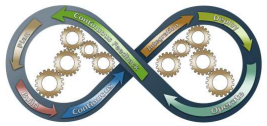
- ❑ Tipos de hipervisores Una definición sencilla de hipervisor podría ser: la parte de la nube privada que gestiona las máquinas virtuales, es decir, es la parte (programa) que permite que múltiples sistemas operativos compartan el mismo hardware. Cada sistema operativo podría usar todo el hardware (procesador, memoria) si no hay otro sistema operativo encendido. Ese es el hardware máximo disponible para un sistema operativo en la nube. Sin embargo, el hipervisor es el que controla y asigna qué parte de los recursos de hardware debe obtener cada sistema operativo, para que cada uno obtenga lo que necesita y no se interrumpa entre sí.

- Hay dos tipos de hipervisores:

  - ✓ Hipervisor de tipo 1: Los hipervisores se ejecutan directamente en el hardware del sistema: un hipervisor integrado 'básico'.

  - ✓ Hipervisor tipo 2: Los hipervisores se ejecutan en un sistema operativo host que proporciona servicios de virtualización, como soporte de dispositivos de E / S y administración de memoria.





- Hipervisores tipo 1:

- ✓ VMware ESX y ESXi Estos hipervisores ofrecen funciones avanzadas y escalabilidad , pero requieren licencia, por lo que los costos son más altos. Su producto vSphere / ESXi está disponible en una edición gratuita y 5 ediciones comerciales.

- ✓ Microsoft Hyper-V El hipervisor de Microsoft, Hyper-V, no ofrece muchas de las funciones avanzadas que ofrecen los productos de VMware. Sin embargo, con XenServer y vSphere, Hyper-V es uno de los 3 principales hipervisores tipo 1. Actualmente, las nuevas versiones de supervisores de Microsoft están íntimamente relacionadas a sus productos de cloud y se les conoce como ‘Azure Stack’.

- ✓ Citrix XenServer Comenzó como un proyecto de código abierto. La tecnología principal del hipervisor es gratuita, pero al igual que ESXi gratuito de VMware, casi no tiene características avanzadas. Xen es un hipervisor de tipo desnudo de tipo 1 y así como Red Hat Enterprise Virtualization usa KVM, Citrix usa Xen en el XenServer comercial.

- ✓ Oracle VM El hipervisor Oracle se basa en el código abierto Xen. Sin embargo, si necesita soporte de hipervisor y actualizaciones de productos, le costará. Oracle VM carece de muchas de las características avanzadas que se encuentran en otros hipervisores de virtualización de metal desnudo.

- Hipervisores tipo 2:

- ✓ VMware Workstation / Fusion / Player VMware Player es un hipervisor de virtualización gratuito. Está destinado a ejecutar solo una máquina virtual (VM) y no permite crear máquinas virtuales. VMware Workstation es un hipervisor más robusto con algunas características avanzadas, como grabación, reproducción y compatibilidad con instantáneas de VM. VMware Workstation tiene tres casos de uso principales: 1. Para ejecutar múltiples sistemas operativos. 2. Para ejecutar versiones diferentes de un sistema operativo en un escritorio. 3. Para desarrolladores que necesitan entornos de sandbox e instantáneas o para laboratorios y con fines de demostración.

- ✓ Servidor VMware VMware Server es un hipervisor de virtualización alojado gratuito que es muy similar a VMware Workstation. VMware ha detenido el desarrollo en el servidor desde 2009.

- ✓ Microsoft Virtual PC Esta es la última versión de Microsoft de esta tecnología de hipervisor, Windows Virtual PC y solo se ejecuta en Windows 7 y solo es compatible con los sistemas operativos Windows que se ejecutan en él.

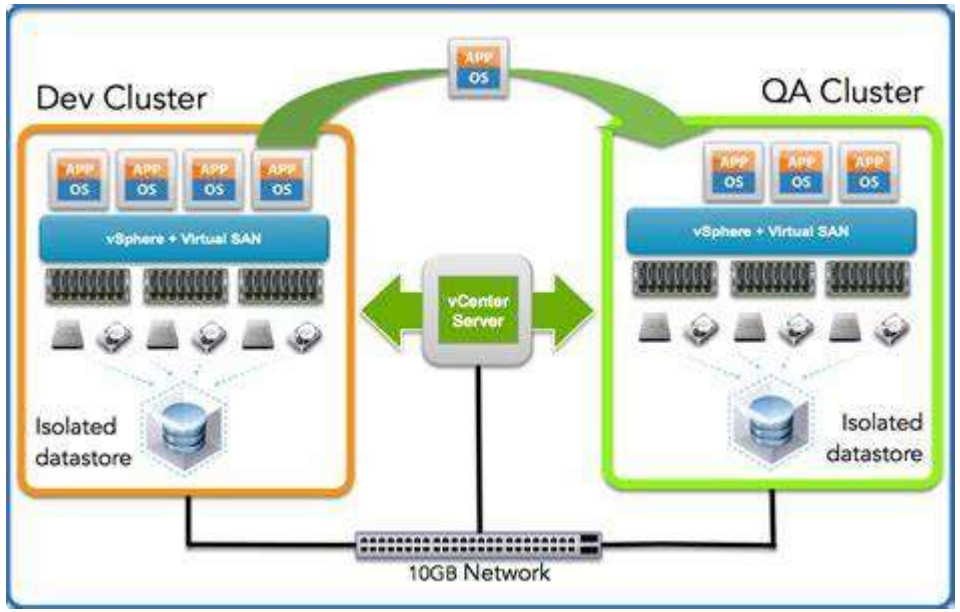
- ✓ Oracle VM VirtualBox La tecnología de hipervisor VirtualBox proporciona un rendimiento y características razonables si desea virtualizar con un presupuesto limitado. A pesar de ser un producto alojado gratuito con una huella muy pequeña, VirtualBox comparte muchas características con VMware vSphere y Microsoft Hyper-V.

- Hipervisores tipo 2:

- ✓ Red Hat Enterprise Virtualization La máquina virtual basada en el kernel (KVM) de Red Hat tiene cualidades tanto de un hipervisor de virtualización alojado como virtual. Puede convertir el núcleo de Linux en un hipervisor para que las máquinas virtuales tengan acceso directo al hardware físico.

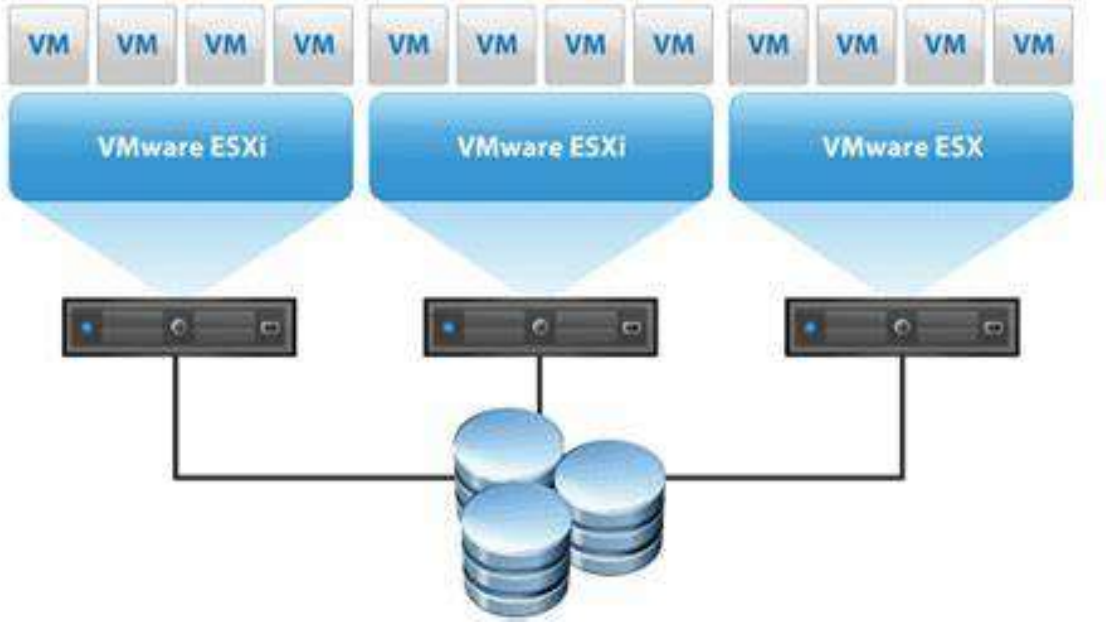
- ✓ KVM Esta es una infraestructura de virtualización para el kernel de Linux. Admite la virtualización nativa en procesadores con extensiones de virtualización de hardware. El KVM de código abierto (o máquina virtual basada en el núcleo) es un hipervisor tipo 1 basado en Linux que se puede agregar a la mayoría de los sistemas operativos Linux, incluidos Ubuntu, Debian, SUSE y Red Hat Enterprise Linux, pero también Solaris y Windows.

- Alta disponibilidad (HA): extiende el concepto de conmutación por error al incorporar un servidor de hardware adicional. Es decir, que en el caso de que la máquina virtual fallase, esta no se iniciaría en la misma pieza de hardware, sino que se iniciaría en un servidor diferente, evitando así el problema de conmutación por error de virtualización por adición del hardware:



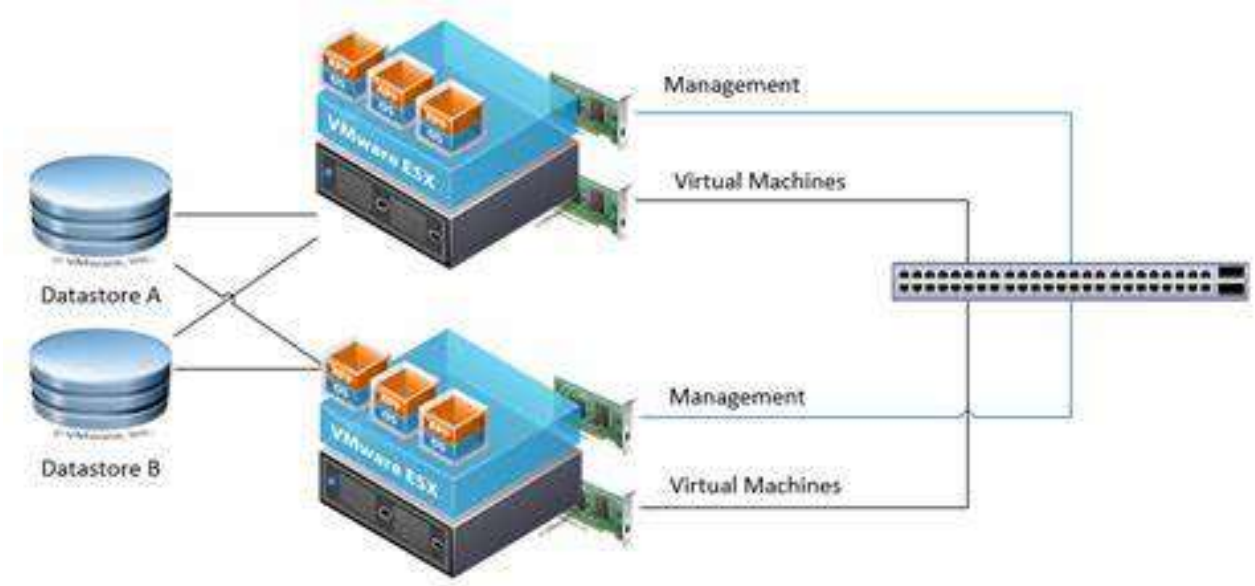
- Agrupamiento (clustering) El agrupamiento está diseñado para garantizar que no se pierdan datos en caso de que haya un fallo de software o de hardware. Hay un gasto extra que se produce por la necesidad de contar con hardware adicional, con el sistema en espejo en espera (en modo standby.

- El sistema SB está listo para asumir el control si fallase el sistema primario; sin embargo, si en el sistema se operan transacciones de millones de euros, mantener un servidor redundante que esté listo para operar cuando haya un fallo, puede ser una inversión que valga la pena.



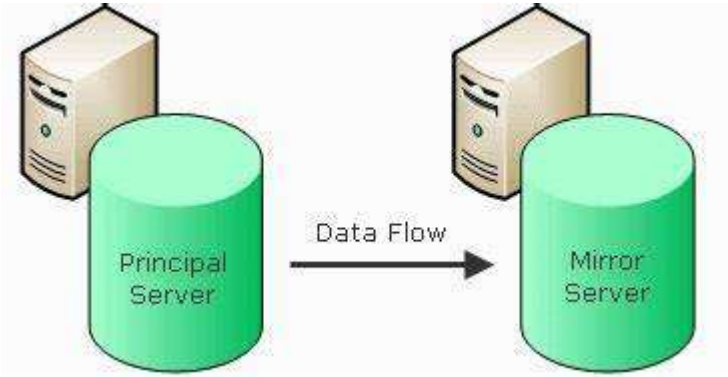
- Esencialmente, el software de coordinación de virtualización ejecuta dos máquinas virtuales en máquinas separadas. Las máquinas virtuales son idénticas en cuanto al sistema operativo y la configuración de la aplicación, pero difieren, naturalmente, en los detalles de sus conexiones de red y hardware local. El supervisor de virtualización se comunica constantemente con las máquinas virtuales en el clúster para confirmar que están trabajando (heartbeat).

- Una VM (máquina virtual) es el servidor primario y es el sistema con el que los usuarios interactúan. La segunda VM sirve como backup (copia de seguridad), lista para actuar en caso de que el servidor primario se caiga:

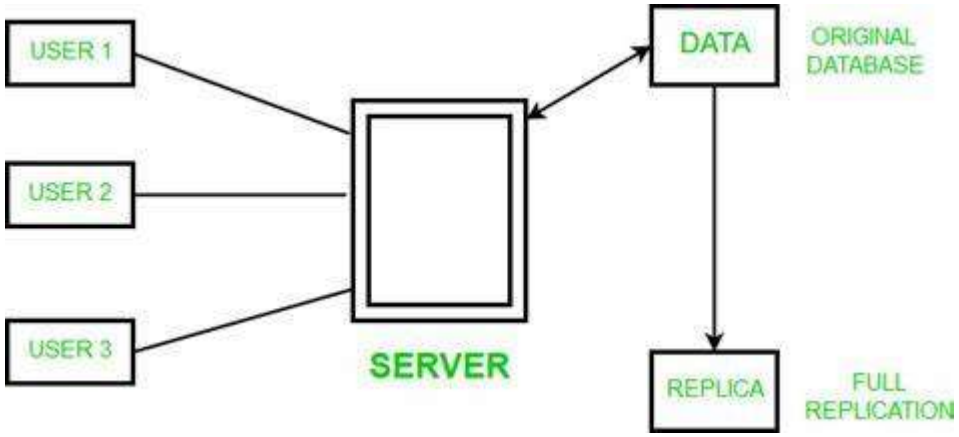


- **Duplicación de datos (data mirroring):** Una forma de mantener los datos disponibles es a través de la duplicación. Como el nombre implica, esta duplicación o reflejo significa que los datos existentes en un sitio son reflejados en otro y ambas contienen la misma información. La duplicación permite la consistencia en tiempo real entre dos fuentes de datos.

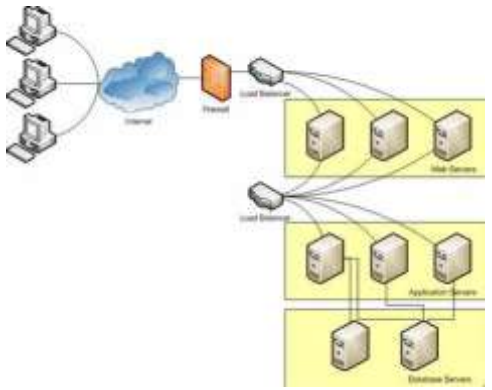
- Esto posibilita el cambio inmediato entre un sistema y otro, es decir, conectando el segundo sistema a la duplicación o el reflejo de los datos del sistema primario:



- **Réplica de datos:** La replicación es otro servicio orientado a mejorar la calidad del servicio de datos. A diferencia de la duplicación (data mirroring) que se enfoca en cómo mantener copias de datos consistentes en tiempo real, la replicación aborda la necesidad de mantener copias completas de los datos para que puedan ser utilizados en la reconstrucción del sistema. Esto se logra enviando copias de datos a un almacenamiento centralizado, lo que permite a una organización tener la seguridad de que en caso de que necesite acceder a los datos críticos por algún motivo, estos están almacenados de forma segura y disponibles en caso de ser necesarios. La eficiencia es vital para la replicación, es decir, que no debemos pensar que porque los datos se están moviendo a una ubicación de almacenamiento hay que olvidarse de que los datos deben fluir correctamente. Un software de replicación inteligente mantiene los cambios, minuto a minuto fluyendo a la ubicación central, asegurando así que una organización de TI pueda localizar rápidamente los datos y usarlos para reconstruir el sistema en caso de fallos:



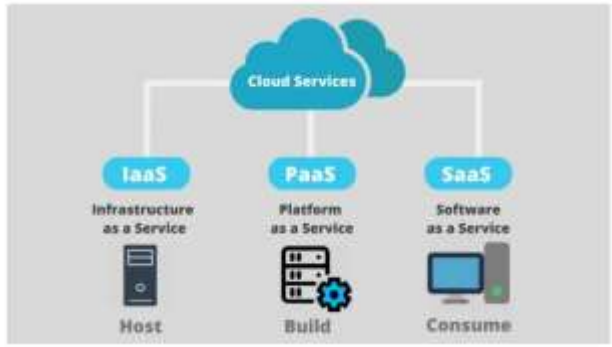
- **Balanceo de carga (load balancing):** El equilibrio o balanceo de carga protege a un sistema de la vulnerabilidad contra cualquier condición de error dada al implementar la denominada redundancia. Esta se logra a través de la ejecución de una o más copias de una máquina virtual en servidores separados. Cuando se ejecutan dos instancias de una máquina virtual y una de ellas se bloquea, la otra continúa funcionando. Si el hardware que da soporte a una de las máquinas virtuales falla, la otra máquina sigue funcionando. De esta manera, se evita que la aplicación sufra una interrupción. El balanceo de carga también hace un mejor uso de los recursos de la máquina. Esto es así porque en lugar de que la segunda máquina virtual esté inactiva y no realice ningún trabajo útil, aunque esté siendo actualizada por la máquina principal, la segunda VM lleva la mitad de la carga y esto hace que al menos la mitad de sus recursos se utilicen. El uso de recursos duplicados puede extenderse más allá de las máquinas virtuales en sí mismas. Las organizaciones que luchan por alcanzar altos niveles de disponibilidad, a menudo, implementan redes duplicadas con cada servidor físico con conexión cruzada con el resto de la red, lo que garantiza que las máquinas virtuales continuarán siendo capaces de comunicarse incluso si parte de la red se cae.



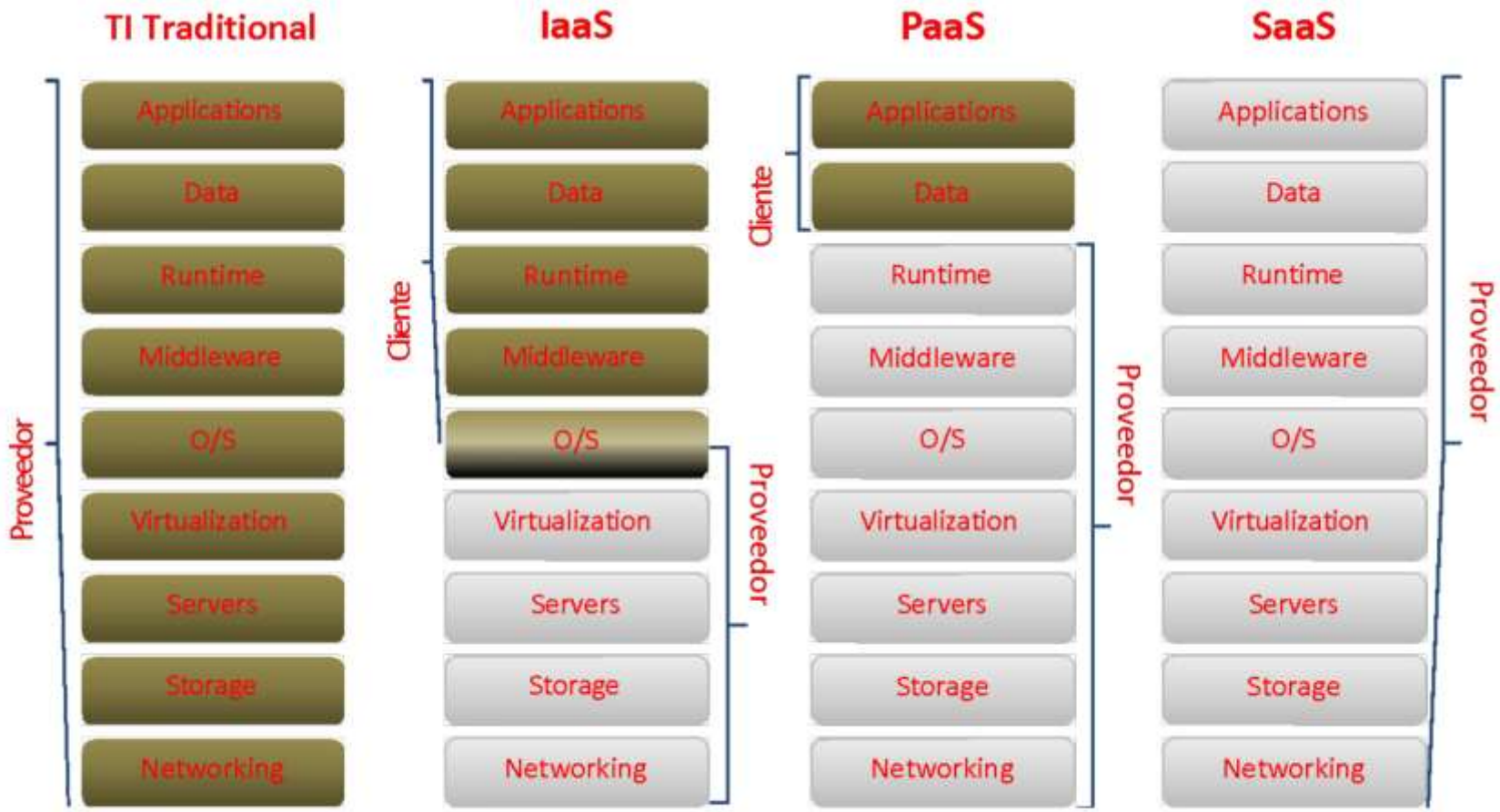
- **Cloud computing:** El paradigma de la nube introduce un cambio en la visualización del sistema y los datos que son propiedad de una empresa. Además, el uso compartido de servicios o recursos, tales como el almacenamiento, hardware y aplicaciones de cloud computing, de una manera totalmente diferente ha facilitado la coherencia de los recursos y las economías de escala a través de su modelo de negocio de pago por uso. Ya no se trata de un conjunto de dispositivos en una ubicación física que ejecutan un programa de software específico con todos los datos y los recursos presentes en un lugar físico, sino que es un sistema que se distribuye geográficamente, involucrando tanto a la aplicación como a los datos:

- Cloud computing:
- Hay 3 tipos de nube El hecho de que la información resida de forma temporal o definitiva en servidores de nube da como resultado que dichos servicios ofrezcan distintos formatos de privacidad que cada usuario puede elegir, según sus necesidades:
  - ✓ Nube pública: Los usuarios acceden a los servicios de manera compartida sin que exista un exhaustivo control sobre la ubicación de la información, que reside en los servidores del proveedor. Es importante resaltar que el hecho de que sean públicas no es un sinónimo de que sean inseguras, pero la realidad es que suelen ser más vulnerables a los ataques. Cuando hablamos de nube pública, queremos decir que toda la infraestructura de computación se encuentra en las instalaciones de una empresa de cloud computing que ofrece el servicio en la nube. La ubicación permanece, por lo tanto, separada del cliente y este no tiene control físico sobre la infraestructura.
  - ✓ Nube privada: Nube privada significa usar una infraestructura en la nube (red) por cada cliente u organización. Si bien no se comparte con otros, se encuentra remotamente localizada. Las empresas tienen la opción de elegir una nube privada en la propia sede, que es más cara, pero tiene la ventaja de que así se puede tener control físico sobre la infraestructura. Resulta evidente que el nivel de seguridad y control es más alto cuando se utiliza una red privada que una red pública. Sin embargo, la reducción de costes puede ser mínima si la empresa necesita invertir en una infraestructura en la nube on premise.
  - ✓ Nube híbrida: Nubes híbridas Combinan características de las dos anteriores, de manera que parte del servicio se puede ofrecer de manera privada (por ejemplo, la infraestructura) y otra parte de manera compartida

- **Infraestructura como servicio (IaaS):** También conocida como HaaS, del inglés hardware as a service, esta arquitectura se basa en el modelo de dotar de forma externalizada a sus usuarios / empresas del hardware necesario. IaaS proporciona hardware, almacenamiento, servidores y espacio de centro de datos o componentes de red. Como inconveniente podemos destacar que se requiere de los mismos conocimientos informáticos en sistemas operativos y redes informáticas que necesitábamos con una arquitectura tradicional.
- **Plataforma como servicio (PaaS):** Se trata de un modelo en el que se proporciona un servicio de plataforma con todo lo necesario para dar soporte al ciclo de diseño, desarrollo y puesta en marcha de aplicaciones y servicios web a través de esta. El proveedor es el encargado de escalar los recursos en caso de que la aplicación lo requiera, de que la plataforma tenga un rendimiento óptimo, de la seguridad de acceso, etc. Para desarrollar software se necesitan bases de datos, herramientas de desarrollo y en ocasiones servidores y redes. Con PaaS, el cliente únicamente se enfoca en desarrollar, depurar y probar ya que las herramientas necesarias para el desarrollo de software son ofrecidas a través de internet, lo que teóricamente permite aumentar la productividad de los equipos de desarrollo gracias a que abstrae del hardware físico al cliente.
- **Software como servicio (SaaS):** Consiste en la entrega de aplicaciones completas como un servicio, permitiendo la abstracción completa, no solo del hardware subyacente, sino incluso de la plataforma, permitiendo al usuario (cliente) dedicarse únicamen



❑ Comparativa de los niveles de servicio en la nube:



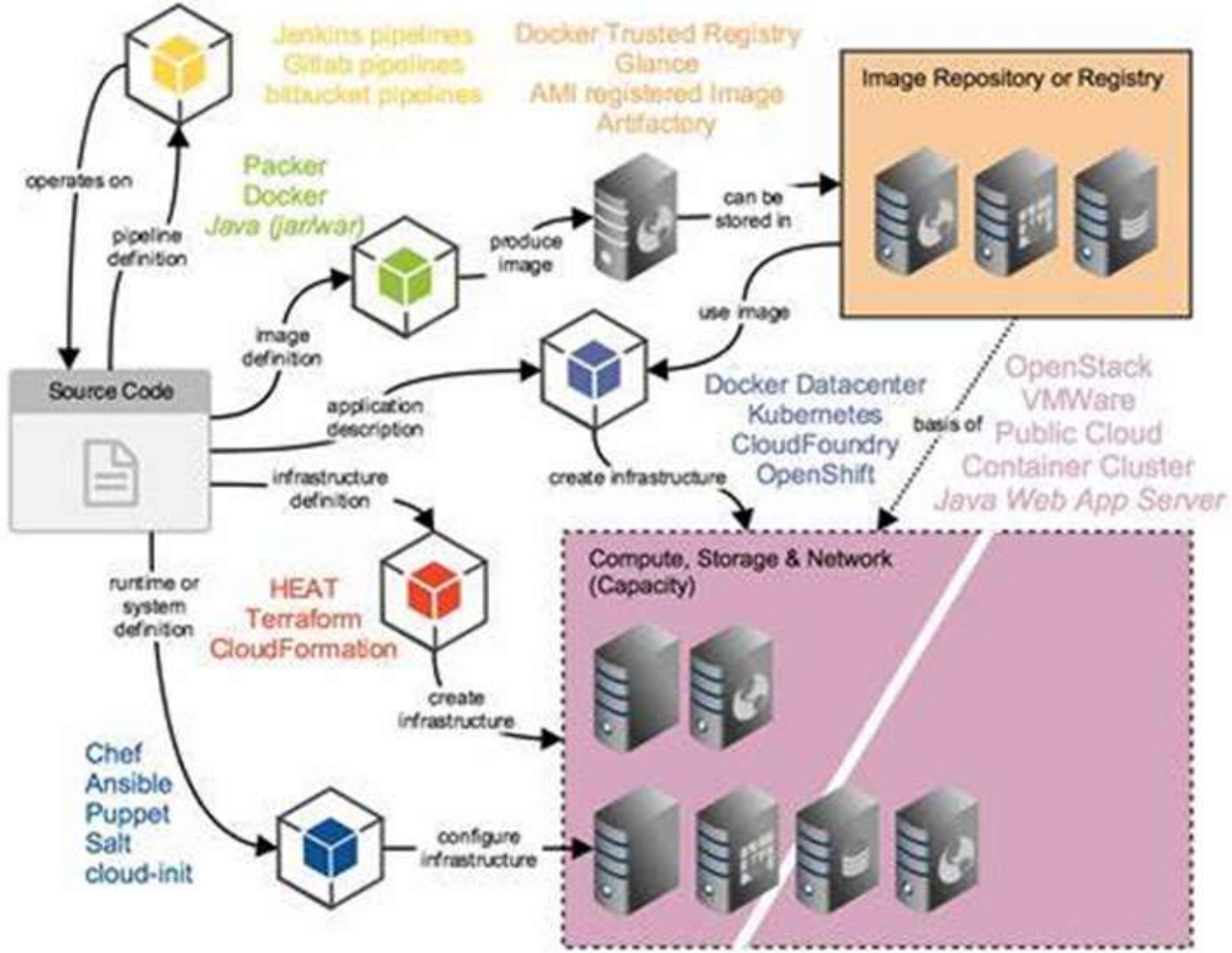
Pie: Comparativa de los niveles de servicio en la nube.

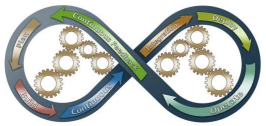
- **Funciones como servicio (FaaS):** Las funciones como servicio son un concepto relativamente moderno, introducido en el mundo cloud por Amazon Web Services con AWS Lambdas en el 2014. Se trata de un tipo de servicio de computación que ejecuta código en respuesta a eventos, pero sin que los usuarios se tengan que preocupar de la infraestructura que sería necesaria para albergar un servicio de estas características: esto permite a los desarrolladores enfocarse únicamente en la creación de código fuente funcional. FaaS es un concepto que se suele asociar, o incluso equiparar, con otro paradigma de computación cloud: serverless.
- **El concepto serverless es el de un PaaS para aplicaciones orientadas a internet o web, en el que el coste es por llamada a la aplicación**. Esto significa que, si no tiene llamadas, el servicio se apaga y queda en espera automáticamente. Y, en caso de tenerlas, el servicio escala automáticamente y de manera transparente para atender la demanda. Pero además, serverless sí que incluye toda la tecnología subyacente para dar servicio a las FaaS, por lo que deben considerarse estas como un subtipo dentro del mundo serverless.

ilas

- **Infraestructura como código (IaC):** La respuesta que ha dado la informática a estos problemas es la optimización y automatización del despliegue de aplicaciones y la configuración de los servidores. Actualmente, es posible configurar servidores a través de la programación y sin intervención de los administradores. Esta nueva forma de administración, que considera a la infraestructura como un aplicativo más a ser gestionado de manera análoga al software, se las conoce como infraestructura programable o infraestructura como código, más conocida como IaC (Huttermann, M., 2012).
- La IaC ofrece las siguientes ventajas frente a la configuración manual tradicional:
  - ✓ Alta eficiencia: Automatiza la mayor parte de la administración de los recursos, lo que lleva a optimizar el ciclo de vida de desarrollo SW.
  - ✓ Reutilización: Una vez se haya descrito una infraestructura como código, esta se puede ejecutar en cualquier momento, todas las veces que se desee, de forma idempotente.
  - ✓ Control de versiones: Al ser código fuente, lo natural es que se almacene en un repositorio, lo que lleva a poder revisar los cambios a lo largo del tiempo.
  - ✓ Minimiza costes y esfuerzo: Al automatizar trabajo tedioso y tendiente a errores.

❑ Infraestructura como código (IaC):





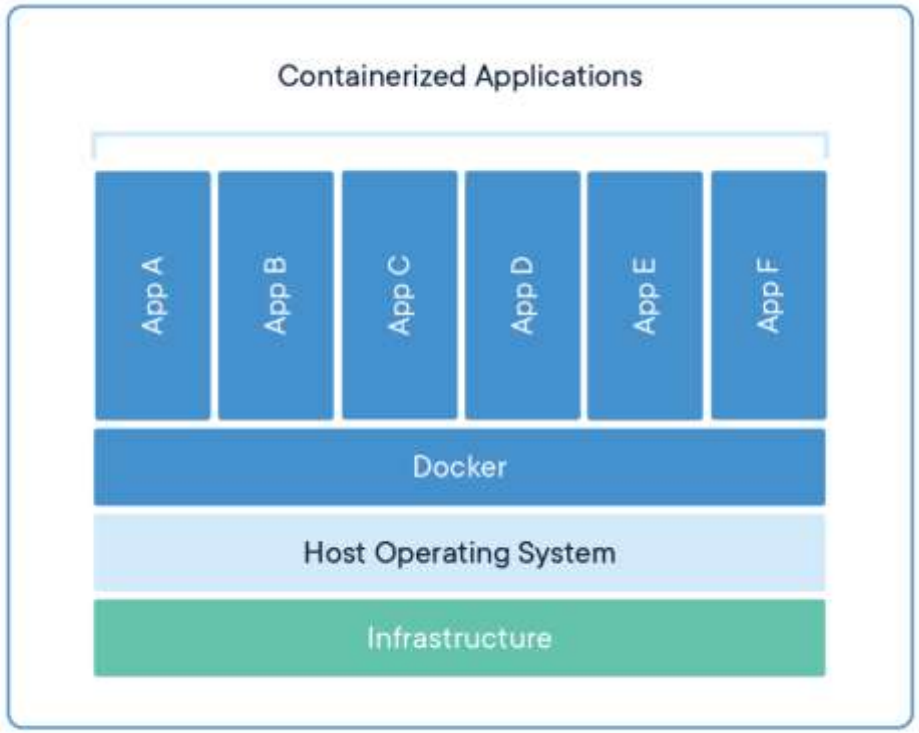
- Existen dos grandes grupos de herramientas de IaC:

1. Herramientas de proveedor Son las herramientas presentes en los grandes proveedores de la nube, y permiten describir y provisionar únicamente sus propias tecnologías. Los tres ejemplos más conocidos son:
   - ✓ **AWS CloudFormation CloudFormation:** es la herramienta interna de IaC de Amazon Web Services (AWS) y, como tal, es prácticamente imprescindible para cualquiera que trabaje con productos de AWS como ELB, S3 o EFS. Utilizarla no conlleva ningún coste adicional, tan solo hay que pagar por los recursos reservados.
   - ✓ **Azure Resource Manager:** Servicio de Azure, plataforma en la nube de Microsoft, que proporciona la administración de infraestructuras mediante plantillas, de forma que implementa y supervisa todos los recursos. Esto da coherencia a la hora de reimplementar recursos ya existentes y permite definir las dependencias de estos.
   - ✓ **Google Cloud Deployment Manager:** Deployment Manager es para la plataforma Google Cloud lo que CloudFormation es para AWS. Con esta herramienta gratuita, los usuarios de recursos de IaaS de Google pueden administrarlos fácilmente mediante archivos de configuración central en el lenguaje de marcado YAML.

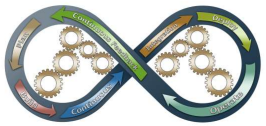
2. Las herramientas multiproveedor más empleadas actualmente son:
   - ✓ **Terraform:** es una herramienta que se utiliza para la construcción, el cambio y el versionado de infraestructura, de manera segura y eficiente. Esta puede administrar tanto servicios existentes como nubes públicas o soluciones internas personalizadas.
   - ✓ **Heat:** Implementa un motor de orquestación para poder lanzar múltiples aplicaciones en la nube basadas en plantillas, proporcionando una compatibilidad total con las plantillas de AWS CloudFormation. Proporciona, a su vez, una API nativa y una API compatible con AWS CloudFormation. Heat es el proyecto más ambicioso de OpenStack.
   - ✓ **Chef Infra:** la solución de IaC de la empresa estadounidense Chef, está disponible desde abril de 2019 bajo la licencia gratuita Apache 2.0 y es utilizado por Facebook, entre otras empresas. Entre las plataformas compatibles se incluyen Google Cloud, Microsoft Azure, Amazon EC2 y OpenStack.
   - ✓ **Puppet:** Herramienta Open Source de gestión desarrollada en Ruby para la administración de sistemas de forma declarativa, la cual al estar basada en modelos no requiere un alto conocimiento de programación para su uso.
   - ✓ **Red Hat Ansible Tower:** La herramienta de infraestructura como código de Ansible forma parte del catálogo de desarrollo de software Red Hat desde el año 2015. Ofrece un panel de control, su propia línea de comandos y una potentísima API REST. En este caso, ambos paquetes disponibles, tanto el estándar como el extendido, son de pago.

2. Las herramientas multiproveedor más empleadas actualmente son:
   - ✓ **Terraform:** es una herramienta que se utiliza para la construcción, el cambio y el versionado de infraestructura, de manera segura y eficiente. Esta puede administrar tanto servicios existentes como nubes públicas o soluciones internas personalizadas.
   - ✓ **Heat:** Implementa un motor de orquestación para poder lanzar múltiples aplicaciones en la nube basadas en plantillas, proporcionando una compatibilidad total con las plantillas de AWS CloudFormation. Proporciona, a su vez, una API nativa y una API compatible con AWS CloudFormation. Heat es el proyecto más ambicioso de OpenStack.
   - ✓ **Chef Infra:** la solución de IaC de la empresa estadounidense Chef, está disponible desde abril de 2019 bajo la licencia gratuita Apache 2.0 y es utilizado por Facebook, entre otras empresas. Entre las plataformas compatibles se incluyen Google Cloud, Microsoft Azure, Amazon EC2 y OpenStack.
   - ✓ **Puppet:** Herramienta Open Source de gestión desarrollada en Ruby para la administración de sistemas de forma declarativa, la cual al estar basada en modelos no requiere un alto conocimiento de programación para su uso.
   - ✓ **Red Hat Ansible Tower:** La herramienta de infraestructura como código de Ansible forma parte del catálogo de desarrollo de software Red Hat desde el año 2015. Ofrece un panel de control, su propia línea de comandos y una potentísima API REST. En este caso, ambos paquetes disponibles, tanto el estándar como el extendido, son de pago.

- **Docker:** Los contenedores tienen una filosofía totalmente diferente a la de las máquinas virtuales. En lugar de tener un sistema operativo completo, los contenedores comparten el mismo kernel o núcleo del sistema operativo host sobre el que se ejecutan:



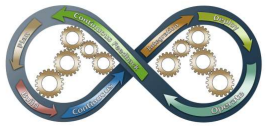
Los contenedores comparten los recursos del host.



- **Docker**

- La plataforma Docker nos permite empaquetar nuestras aplicaciones con todo lo necesario para su ejecución (librerías, código, herramientas, configuraciones), eliminando así dependencias del sistema operativo de la máquina host y facilitando un despliegue muy rápido de nuestras aplicaciones. El motor de Docker será el encargado de desplegar y gestionar los contenedores de nuestras aplicaciones, pero, en lugar de reservar parte de los recursos de hardware de la máquina para cada contenedor, lo que hace es compartirlos entre todos los contenedores, permitiendo optimizar su uso y eliminando la necesidad de tener sistemas operativos separados para conseguir el aislamiento.

- Ventajas de utilizar contenedores En la administración tradicional de servidores de aplicaciones nos encontramos con una serie de tareas que se repiten constantemente en cada máquina: instalación del sistema operativo, actualizaciones y parches necesarios, instalar la última versión de la aplicación y sus dependencias. A medida que aumenta el número de máquinas a gestionar, crece la dificultad de mantenerlas actualizadas con nuevas actualizaciones de seguridad, nuevas versiones de la aplicación, cambios en las dependencias, etc. Se trata de un proceso bastante propenso a errores.



- **Docker**

- Con la llegada de Docker tenemos la posibilidad de empaquetar en una imagen todo el código de nuestra aplicación, su sistema operativo, sus dependencias, binarios, ficheros de configuración (y solamente hacer este proceso una vez). Si alguno de nuestros contenedores falla y termina no haría falta sacar a ese servidor del sistema para reparar el problema, sino que simplemente volveremos a desplegar una nueva instancia del contenedor de nuestra aplicación. De este modo, todas las instancias de la aplicación serían copias idénticas que no haría falta configurar individualmente.

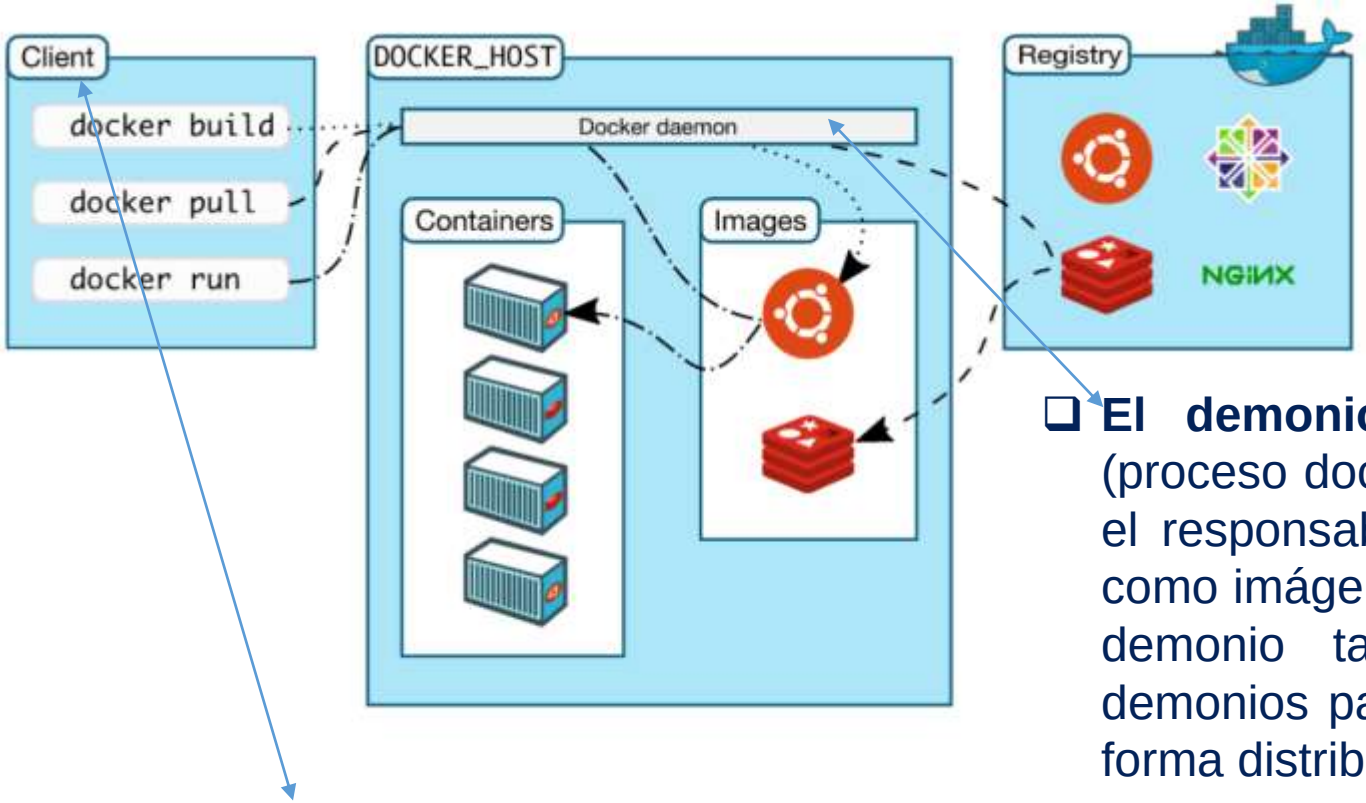
- Podríamos resumir el proceso de esta nueva visión en tres fases:

  - Construir la imagen. Empaquetar consistentemente todo lo que nuestra aplicación necesita para ser ejecutada.

  - Distribuir la imagen. Haremos disponible la imagen para utilizarla en nuestro datacenter, en la nube o en la máquina local del desarrollador.

  - Ejecutar la imagen. Desplegar contenedores a partir de la imagen de una manera rápida, sencilla y consistente.

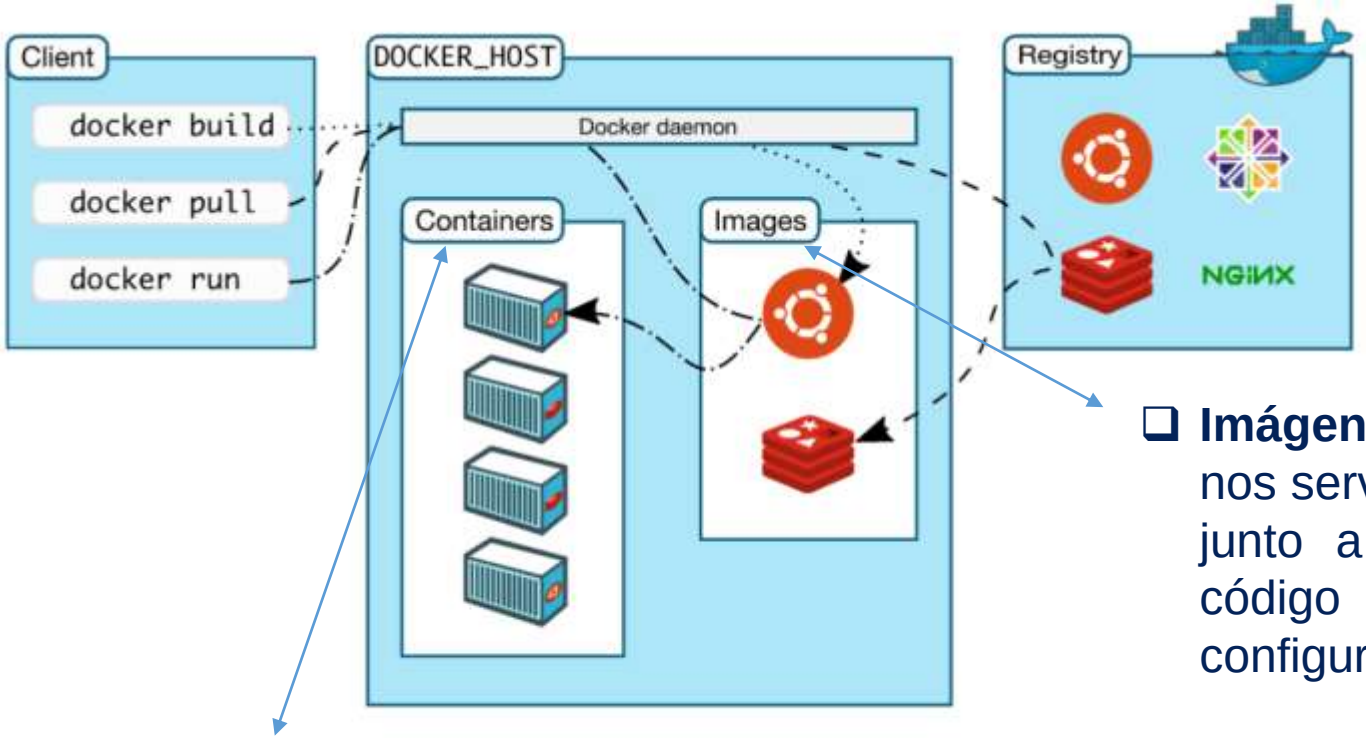
- **Docker:** Arquitectura de Docker La arquitectura de Docker sigue un modelo cliente-servidor. En la siguiente imagen vemos los principales componentes de la arquitectura de Docker.



- **El demonio de Docker:** El demonio de Docker (proceso dockerd) escucha las solicitudes del API y es el responsable de administrar los objetos de Docker, como imágenes, contenedores, redes y volúmenes. Un demonio también puede comunicarse con otros demonios para administrar los servicios de Docker de forma distribuida

- **El cliente Docker:** permite a los usuarios interactuar con el demonio de Docker utilizando la línea de comandos. El cliente Docker y el demonio de Docker se comunican mediante un API REST, ya sea a través de sockets UNIX o mediante una interfaz de red. Ambos no tienen por qué ejecutarse en la misma máquina, ya que el cliente Docker podrá conectarse a un demonio de Docker de un sistema remoto.

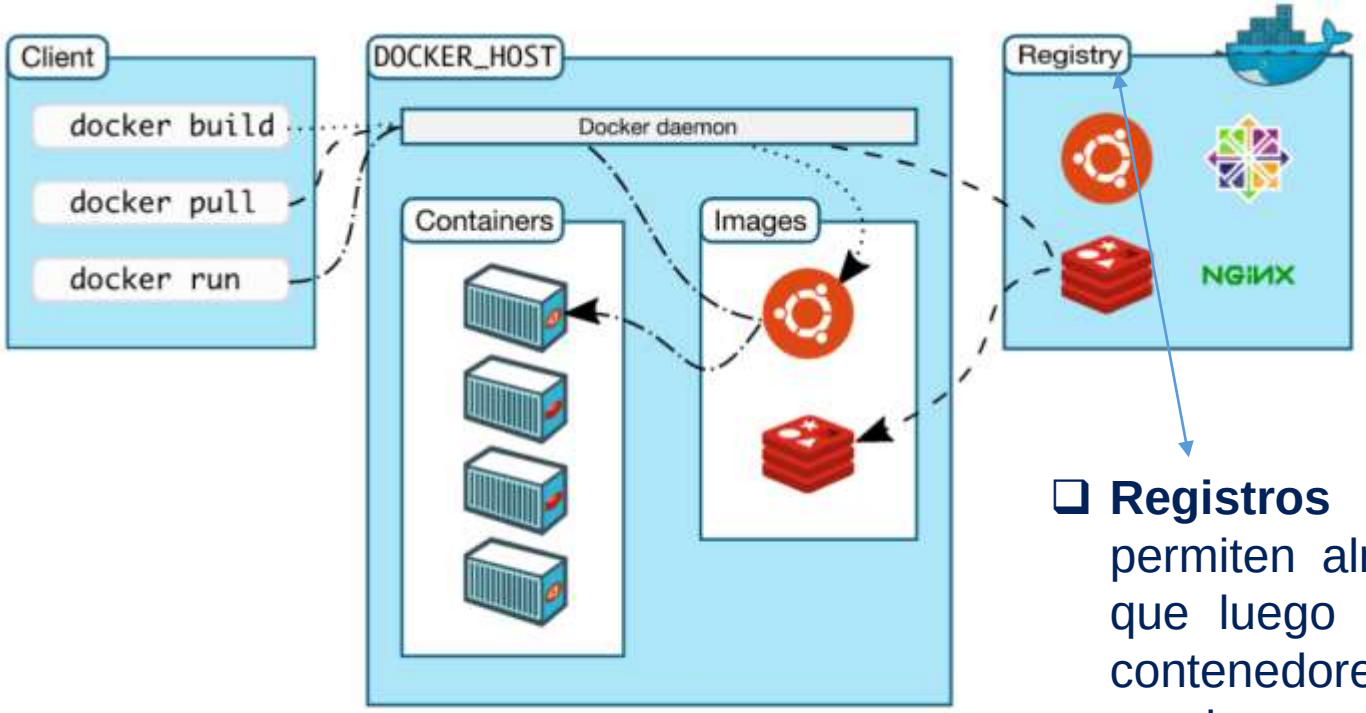
- **Docker:** Arquitectura de Docker La arquitectura de Docker sigue un modelo cliente-servidor. En la siguiente imagen vemos los principales componentes de la arquitectura de Docker.



- **Imágenes:** Las imágenes son objetos de Docker que nos servirán para empaquetar una aplicación o servicio junto a todo lo necesario para su funcionamiento: código de la aplicación, librerías dependientes, configuraciones, etc.

- **Los contenedores:** son instancias o ejecuciones de la imagen de nuestra aplicación o servicio. Los contenedores se comportan como entornos aislados y seguros, por lo que en una misma máquina podemos tener varios contenedores en ejecución de la misma imagen. Para entender mejor la diferencia entre imágenes y contenedores podemos ver su paralelismo con un fichero ejecutable y el proceso que lo ejecuta. La imagen sería el equivalente al ejecutable, y el contenedor similar a los distintos procesos que hay en ejecución del ejecutable.

- **Docker:** Arquitectura de Docker La arquitectura de Docker sigue un modelo cliente-servidor. En la siguiente imagen vemos los principales componentes de la arquitectura de Docker.



- **Registros de Docker:** Los registros de Docker permiten almacenar y distribuir imágenes de Docker que luego serán utilizadas para la creación de los contenedores. Docker Hub es un registro público que puede ser utilizado por los usuarios para compartir imágenes. Por defecto, Docker buscará imágenes en Docker Hub, aunque, como veremos, existen otros repositorios públicos en la nube. Además, podremos disponer de nuestro propio registro de imágenes privado.

- **Espacios de nombres (namespaces):** Docker utiliza una tecnología llamada espacios de nombres o namespaces para proporcionar un espacio de trabajo aislado a los contenedores. Cuando se ejecuta un contenedor, Docker crea un conjunto de espacios de nombres para ese contenedor. Estos espacios de nombres crean una capa de aislamiento. Cada aspecto de un contenedor se ejecuta en un espacio de nombres separado y su acceso está limitado a ese espacio de nombres.

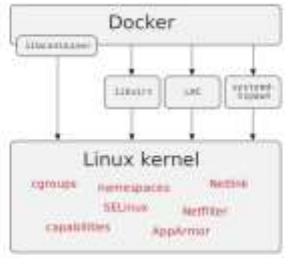
- Algunos **de los namespaces de Linux** utilizados por Docker son:

  - Espacio de nombres pid: aislamiento de los procesos.
  - Espacio de nombres net: gestión de las interfaces de red.
  - Espacio de nombres ipc: gestión del acceso a los recursos de IPC.
  - Espacio de nombres mnt: gestión de puntos de montaje del sistema de archivos.
  - Espacio de nombres uts: Aislamiento del núcleo y los identificadores de versión.

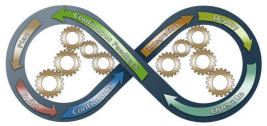
- **Grupos de control (cgroups)** Docker Engine en Linux también se basa en otra tecnología llamada grupos de control (cgroups). Un grupo de control limita una aplicación a un conjunto específico de recursos. Los grupos de control permiten que Docker Engine comparta los recursos de hardware disponibles con los contenedores y, opcionalmente, imponga límites y restricciones. Por ejemplo, puede limitar la memoria disponible para un contenedor específico.

- **Sistemas de archivos de unión (UnionFS):** En Linux, el servicio de sistemas de archivos de unión (o UnionFS) permite montar un sistema de archivos formado por la unión de otros sistemas de archivos. Funciona creando capas, haciéndolos muy livianos y rápidos. Docker Engine utiliza UnionFS para proporcionar los bloques de construcción para contenedores. Docker Engine puede usar múltiples variantes de UnionFS, incluidas AUFS, btrfs, vfs y DeviceMapper.

- **Formato de contenedor Docker:** combina servicios de Linux, como son los espacios de nombres, grupos de control o los sistemas de archivos de unión, en un envoltorio denominado formato contenedor. El formato de contenedor predeterminado de Docker es libcontainer.



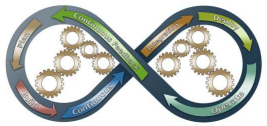
Abcontainer ofrece abstracción para virtualización y aislamiento de contenedores.



- **Registros Docker:** Los registros Docker nos permiten almacenar las imágenes de Docker que vayamos creando de manera que estén disponibles para su descarga, cuando sea necesario, desde nuestros sistemas. Docker por defecto utilizará su registro Docker Hub, que ofrece repositorios tanto públicos como privados. Sin embargo, Docker Hub no es el único registro disponible públicamente, aunque sí el más popular. Existen otros registros con sus propias características y funcionalidades. Según nuestras necesidades podemos decantarnos por uno u otro atendiendo al coste, rendimiento, seguridad, geo-replicación, etc. Además, también podríamos crearnos un registro privado en nuestro datacenter o nuestra cloud privada y conseguir lo mismo pero con el añadido de la seguridad para la organización: https://hub.docker.com/

```
$ docker login -u <user> -p <password> url.to/my_registry

$ docker push raulsalgado/mirepositorio:tagname
```

```
root@raul-virtual-machine:~# docker login -u raulsalgado -p Raul2025@  url.to/mirepositorio
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```



- **Ejecución de imágenes:** Una vez sabemos qué imagen de Docker queremos ejecutar, tenemos varias opciones:

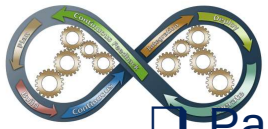
  - docker pull si tan solo queremos descargarla para ejecutarla después o para poder usarla en una construcción.

  - docker run si queremos ejecutar un comando disponible dentro de la imagen.

- Si por el contrario existe uno o más contenedores ejecutándose, y queremos meternos en uno de ellos para inspeccionar el estado, podemos usar docker exec para hacerlo. Vamos a ver todo esto con un ejemplo en el que se usarán las opciones de configuración más comunes de estos comandos, que se pueden consultar en la web de Docker o usando el modificador --help de cada comando docker empleado.

- Para ver los contaniers que corren:

```
root@raul-virtual-machine:~/docker# sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```



- Para ver las imágenes que tengo :

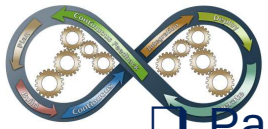
```
root@raul-virtual-machine:~/docker# sudo docker image ls

REPOSITORY              TAG       IMAGE ID       CREATED        SIZE
kalilinux/kali-rolling  latest    7bf1d2b8070b   12 days ago    116MB
nginx                   latest    a99a39d070bf   3 weeks ago    142MB
nginxdemos/hello        latest    cda867cb0287   3 weeks ago    40.7MB
ubuntu                  latest    6b7dfa7e8fdb   8 weeks ago    77.8MB
odoo                    latest    8c1aad3d422f   4 months ago   1.53GB
postgres                13        e30fb19f297b   4 months ago   373MB
gns3/kalilinux          latest    3d6c09a18884   5 years ago    2.29GB
```

- Ejecuto con detache el demonio en segundo plano:

```
root@raul-virtual-machine:~/docker# docker run -d cda867cb0287
819c69d8adf9ce6ce23c12e98e612c35bf91e21a05e99a61ae236606544420dc
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS     NAMES
819c69d8adf9   cda867cb0287   "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   80/tcp    relaxed_benz
```

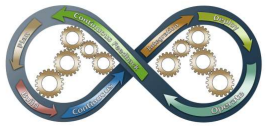
- Parar un contenedor vamos al container_id:

```
root@raul-virtual-machine:~/docker# docker stop 819c69d8adf9
```



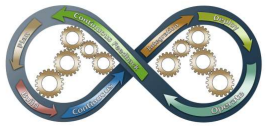
- Para ver las imágenes que tengo y ejecutar e Docker de forma interactiva:

```
root@raul-virtual-machine:~/docker# docker image ls
REPOSITORY              TAG       IMAGE ID       CREATED        SIZE
kalilinux/kali-rolling  latest    7bf1d2b8070b   12 days ago    116MB
nginx                   latest    a99a39d070bf   3 weeks ago    142MB
nginxdemos/hello        latest    cda867cb0287   3 weeks ago    40.7MB
ubuntu                  latest    6b7dfa7e8fdb   8 weeks ago    77.8MB
odoo                    latest    8c1aad3d422f   4 months ago   1.53GB
postgres                13        e30fb19f297b   4 months ago   373MB
gns3/kalilinux          latest    3d6c09a18884   5 years ago    2.29GB
root@raul-virtual-machine:~/docker# docker run -it nginxdemos/hello
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: /etc/nginx/conf.d/default.conf is not a file or does not exist
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/02/03 14:52:38 [notice] 1#1: using the "epoll" event method
2023/02/03 14:52:38 [notice] 1#1: nginx/1.23.3
2023/02/03 14:52:38 [notice] 1#1: built by gcc 12.2.1 20220924 (Alpine 12.2.1_git20220924-r4)
2023/02/03 14:52:38 [notice] 1#1: OS: Linux 5.15.0-58-generic
2023/02/03 14:52:38 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1024:524288
2023/02/03 14:52:38 [notice] 1#1: start worker processes
2023/02/03 14:52:38 [notice] 1#1: start worker process 21
2023/02/03 14:52:38 [notice] 1#1: start worker process 22
```



❑ Para ejecutar e Docker : **root@raul-virtual-machine:~/docker# docker pull nginxdemos/hello**

Using default tag: latest
latest: Pulling from nginxdemos/hello
Digest: sha256:409564c3a1d194fcfd85cad7a231b61670ab6c40d04d80e86649d1fe7740436e
Status: Image is up to date for nginxdemos/hello:latest
docker.io/nginxdemos/hello:latest
root@raul-virtual-machine:~/docker# docker run -P -d nginxdemos/hello
0becb3c629b5c8b7bc653fc4216e440cadd145b84dda943e0af29b63ce5757d9
root@raul-virtual-machine:~/docker#
root@raul-virtual-machine:~/docker# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0becb3c629b5 nginxdemos/hello "/docker-entrypoint.…" About a minute ago Up About a minute 0.0.0.0:49153->80/tcp, :::49153->80/tcp epic_chaum
819c69d8adf9 cda867cb0287 "/docker-entrypoint.…" 19 minutes ago Up 19 minutes 80/tcp relaxed_benz
root@raul-virtual-machine:~/docker# curl 127.0.0.1:49153
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>

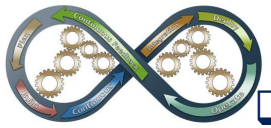


- Montar sistema de ficheros local y ejecutar un server Ubuntu : Docker solo es capaz de montar rutas absolutas, por lo que, si queremos montar una carpeta dentro del directorio donde estamos, debemos anteponer el path. Una manera sencilla con bash sería así:

```
root@raul-virtual-machine:~/docker# mkdir scr
root@raul-virtual-machine:~/docker# ls -all
total 12
drwxr-xr-x 3 root root 4096 feb  3 16:11 .
drwx------ 8 root root 4096 feb  3 15:37 ..
drwxr-xr-x 2 root root 4096 feb  3 16:11 scr
root@raul-virtual-machine:~/docker# ls scr/
$ docker run -ti -v $(pwd)/src:/code ubuntu:latest
root@raul-virtual-machine:~/docker# docker run -ti -v $(pwd)/src:/code ubuntu:latest
root@921ce7c9ffcb:/#
```

- Salgo de la terminal interactiva de Ubuntu:

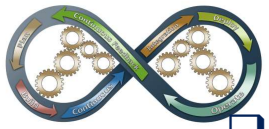
```
root@921ce7c9ffcb:/# exit
exit
root@raul-virtual-machine:~/docker#
```



- Actualizo paquetes:

root@raul-virtual-machine:~/docker# docker run -ti -v $(pwd)/src:/code ubuntu:latest
root@702dd2d7f4f4:/# apt-get update

Get:1 http://archive.ubuntu.com/ubuntu jammy InRelease [270 kB]
Get:2 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [114 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [107 kB]
Get:5 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages [798 kB]
Get:6 http://security.ubuntu.com/ubuntu jammy-security/restricted amd64 Packages [681 kB]
Get:7 http://archive.ubuntu.com/ubuntu jammy/main amd64 Packages [1792 kB]
Get:8 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [754 kB]
Get:9 http://security.ubuntu.com/ubuntu jammy-security/multiverse amd64 Packages [4732 B]
Get:10 http://archive.ubuntu.com/ubuntu jammy/restricted amd64 Packages [164 kB]
Get:11 http://archive.ubuntu.com/ubuntu jammy/multiverse amd64 Packages [266 kB]
Get:12 http://archive.ubuntu.com/ubuntu jammy/universe amd64 Packages [17.5 MB]
Get:13 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [1078 kB]
Get:14 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages [730 kB]
Get:15 http://archive.ubuntu.com/ubuntu jammy-updates/multiverse amd64 Packages [8978 B]
Get:16 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 Packages [1008 kB]
Get:17 http://archive.ubuntu.com/ubuntu jammy-backports/main amd64 Packages [49.0 kB]
Get:18 http://archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages [22.4 kB]
Fetched 25.4 MB in 4s (6734 kB/s)
Reading package lists... Done



❑ Instalo paquetes:

root@702dd2d7f4f4:/# # apt-get install nmap

root@702dd2d7f4f4:/# exit

❑ Commiteo el paquete fuera del bash que antes he hecho el exit:

root@raul-virtual-machine:~/docker# docker commit 702dd2d7f4f4 myapp:version2

sha256:22a6aaa9fbcf59d880554616cf620130e309492d5d7043b9d6ab92c365d3dc09

root@raul-virtual-machine:~/docker# docker images myapp

REPOSITORY TAG IMAGE ID CREATED SIZE

myapp version2 22a6aaa9fbcf 2 minutes ago 119MB

root@raul-virtual-machine:~/docker# docker run -it myapp:version2 /bin/bash

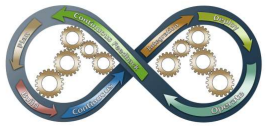
root@f26adb225652:/#

❑ Los parámetros ‘-it’ casi siempre van juntos, pero en realidad son dos distintos:

➢ -i especifica que la sesión va a ser interactiva, dejando STDIN abierto.

➢ -t (--tty) prepara una sesión TTY dentro del contenedor que lee de STDIN.

➢ Entre las dos, se consigue de manera efectiva el poder enviarle comandos al contenedor

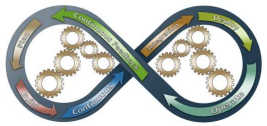


- **Registro Docker privado:** Los registros privados de contenedores nos permitirán crear y publicar imágenes en **un host al que solo pueda acceder nuestro equipo, empresa o clientes**. **Docker nos proporciona la imagen registry para crearnos nuestro propio registro**. Esto nos permite controlar el acceso al mismo mediante mecanismos de autenticación propios, certificados SSL privados o, incluso, usar un backend de storage personalizado dentro de nuestra cloud de elección (por ejemplo, un bucket de AWS S3). **En el caso de servicios como Kubernetes, un registry privado es la única manera de utilizar imágenes propias dentro del clúster.**

- **El siguiente comando crea un Docker Registry privado en un contenedor:**

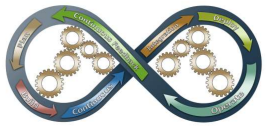
  - Accesible a través del puerto 5000 del host.
  - Con almacenamiento un directorio local.
  - Disponible como servicio, siempre levantado mientras el demonio Docker esté funcionando

```
root@f26adb225652:/#  docker run -d -p 5000:5000 -v $HOME/registry:/var/lib/registry \
>
```



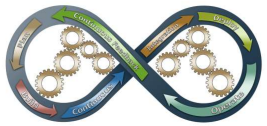
- **Volúmenes de Docker:** Volúmenes Como hemos visto, es posible montar uno o más directorios locales a los contenedores: esto es lo que se conoce como un volumen de tipo bind mount:

```
root@raul-virtual-machine:~/docker# docker ps -a
CONTAINER ID   IMAGE              COMMAND                  CREATED          STATUS                      PORTS                                       NAMES
3baefc10f8d5   docker             "dockerd-entrypoint.…"   2 minutes ago    Created                                                         vibrant_cohen
f26adb225652   myapp:version2     "/bin/bash"              15 minutes ago   Exited (127) 3 minutes ago                                      vigilant_swirles
702dd2d7f4f4   ubuntu:latest      "bash"                   23 minutes ago   Exited (127) 18 minutes ago                                     busy_herschel
921ce7c9ffcb   ubuntu:latest      "bash"                   28 minutes ago   Exited (127) 24 minutes ago                                     heuristic_solomon
0becb3c629b5   nginxdemos/hello   "/docker-entrypoint.…"   39 minutes ago   Up 39 minutes               0.0.0.0:49153->80/tcp, :::49153->80/tcp   epic_chaum
af6a44029385   nginxdemos/hello   "/docker-entrypoint.…"   48 minutes ago   Exited (0) 41 minutes ago                                       sweet_babbage
819c69d8adf9   cda867cb0287       "/docker-entrypoint.…"   56 minutes ago   Up 56 minutes               80/tcp                                      relaxed_benz
```



- **Volúmenes de Docker:** Volúmenes Como hemos visto, es posible montar uno o más directorios locales a los contenedores: esto es lo que se conoce como un volumen de tipo bind mount:

```
root@raul-virtual-machine:~/docker# snap install jq
jq 1.5+dfsg-1 from Michael Vogt (mvo✪) installed
root@raul-virtual-machine:~/docker# docker inspect 819c69d8adf9 | jq .[0].Mounts
[]
root@raul-virtual-machine:~/docker#
root@raul-virtual-machine:/home/raul/Desktop# jq --version
jq-1.6

root@raul-virtual-machine:~/docker# docker volume create mis-datos
mis-datos
root@raul-virtual-machine:~/docker#



root@raul-virtual-machine:~/docker# docker volume ls
DRIVER    VOLUME NAME
local     mis-datos
```



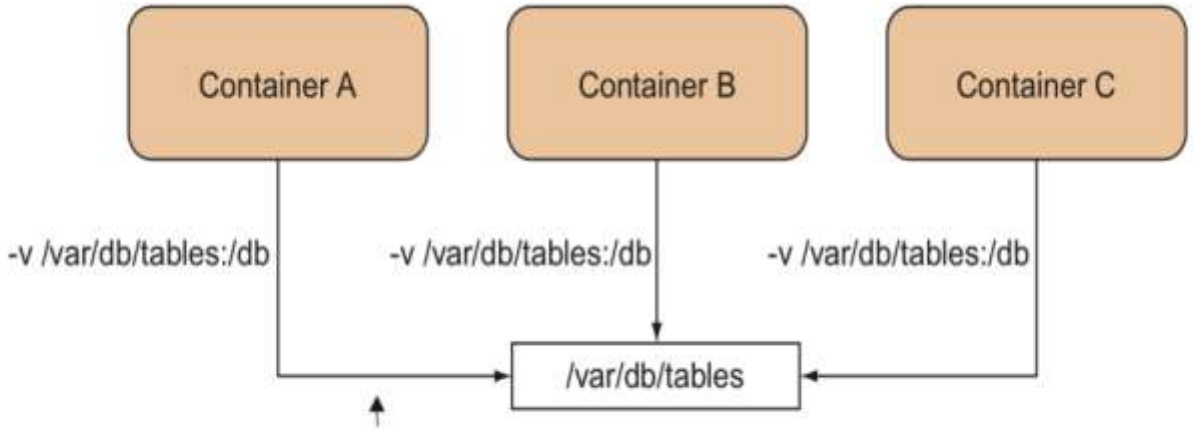
❑ Volúmenes de Docker:

root@raul-virtual-machine:~/docker# **docker volume inspect mis-datos => JSON DEL DOCKER VOLUME CREADO**

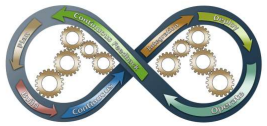

```
[
    {
        "CreatedAt": "2023-02-03T16:44:27+01:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/snap/docker/common/var-lib-docker/volumes/mis-datos/_data",
        "Name": "mis-datos",
        "Options": {},
        "Scope": "local"
    }
]
```

- **Contenedores de datos:** Cuando utilizamos muchos volúmenes de datos puede ser complicado gestionar el arranque de los contenedores. En estos casos podemos utilizar el patrón de diseño de contenedor de datos, que nos permitirá simplificar la gestión de nuestros volúmenes de datos. Un contenedor de datos será un contenedor que solamente tendrá volúmenes montados y no ejecutará ninguna aplicación. De hecho, no es necesario que el contenedor esté en ejecución para poder montar sus volúmenes en otros contenedores, puede estar detenido perfectamente.

- En la siguiente imagen muestra cómo montaríamos el mismo volumen en varios contenedores sin utilizar un contenedor de datos. Vemos que es necesario indicar la ruta en cada uno de ellos:



Contenedores montando el mismo directorio del host.



❑ Contenedores de datos:

$ docker volume create mis-datos mis-datos
$ docker volume ls
DRIVER
VOLUME NAME
local mis-datos
$ docker volume inspect mis-datos
[
{ "CreatedAt": "2020-07-12T09:15:03Z",
"Driver": "local",
"Labels": {},
"Mountpoint": "/var/lib/docker/volumes/mis-datos/_data",
"Name": "mis-datos", "Options": {},
"Scope": "local" }
]

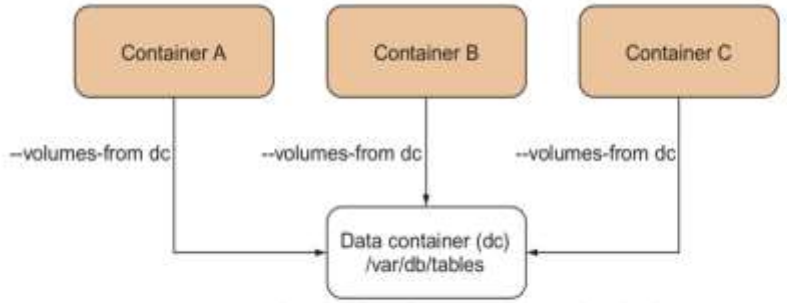
- **Utilizando volúmenes de datos:** Cuando un contenedor monta los volúmenes no necesitará saber dónde se encuentran los datos en el disco. Solamente necesitará saber el nombre del contenedor de datos, lo que lo hace mucho más portable.
- Para crear nuestro contenedor de datos, lo habitual es utilizar como imagen base busybox: se trata de una imagen muy reducida, con funcionalidades básicas de Linux, pero más ligera que alpine. Ya que no vamos a ejecutar nada en el contenedor, nos será suficiente:

$docker run -v /datos-compartidos --name contenedorDatos busybox \ touch /datos-compartidos/ficheroVacio

- Si queremos copiar archivos en el contenedor de datos tendremos que utilizar el comando docker cp, ya que no es accesible directamente desde el host.
- El siguiente comando copiaría un fichero de configuración en el directorio /config del contenedor de datos:

$ docker cp config.conf contenedorDatos:/config/

- Una vez creado y configurado nuestro contenedor de datos, montaremos sus volúmenes al ejecutar otros contenedores con la opción –volumes-from, indicando el identificador o nombre del contenedor de datos:



Contenedores enlazados a un contenedor de datos.

$ docker run -it --name ContenedorA \ --volumes-from contenedorDatos busybox /bin/sh / # ls /datos-compartidos ficheroVacio

- **Importar y exportar contenedores de datos:** Cuando utilizamos contenedores de datos, el proceso de moverlos de una máquina a otra se simplifica muchísimo. Podemos exportarlos a un fichero TAR y después importarlo en otra máquina:

  $ docker export contenedorDatos > contenedorDatosExport.tar
  $ docker import contenedorDatosExport.tar

- **Redes en Docker:** Los contenedores, además de compartir almacenamiento por medio de volúmenes, también pueden relacionarse unos con otros compartiendo redes.
- Veremos los distintos tipos de redes disponibles en Docker y cuáles son sus principales usos y funcionalidades:
  - Bridge: es el controlador por defecto. Permite la comunicación entre dos contenedores independientes ejecutándose en el mismo anfitrión o host. Los contenedores creados se asociarán por defecto a la red existente llamada bridge. Pero podremos crear tantas redes de este tipo como necesitemos.
  - Host: permite eliminar el aislamiento de red entre el contenedor y el anfitrión.
  - Overlay: nos permitirá conectar contenedores y/o servicios de **Docker Swarm** corriendo en diferentes demonios Docker, es decir, en distintos nodos.
  - Macvlan: nos permite asignar una dirección MAC a un contenedor, permitiendo así redirigir directamente tráfico de red hacia ellos.
  - None: deshabilita todas las redes del contenedor.
  - Si listamos las redes iniciales que tenemos tras la instalación de Docker, veremos solamente tres redes: la red por defecto de tipo bridge, la red que utilizará un contenedor con el controlador host y, por último, la red none que simplemente tiene la interfaz de bucle invertido:

❑ Redes en Docker:

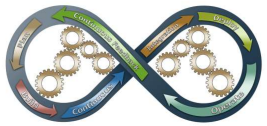
$docker network ls

NETWORK ID	NAME	DRIVER	SCOPE
1620617a10d6	bridge	bridge	local
707918bf2b62	host	host	local
f12a0ea3db2b	none	null	local

$ docker run -dit --rm --name contenedorA alpine sh
$ docker run -dit --rm --name contenedorB alpine sh

❑ Si inspeccionamos la red bridge veremos que tiene asociados nuestros dos contenedores, cada uno en su propia subred:

```
$ docker network inspect bridge | jq .[0].Containers
{
  "55622a13407b3f4d7f144c2e3287e971a476ea7d8281185e96624a84c58b1815": {
    "Name": "contenedorB",
    "EndpointID": "e335702f2028c8fbea45cd00b2e63487eafb8c7e6d84831c2985191d536c9681",
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
  },
  "bb017f0c56e9fa59cfce5ef166337120115c4e0adb4fd1c6e2d1782d7bf37b32": {
    "Name": "contenedorA",
    "EndpointID": "e0af55f5b7b199f1b22d0cb215d653cd373b55f57ed133ef925e5f5e18182885",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
}
```



- **Redes en Docker:** Enlazando contenedores directamente Ahora vamos a recrear el contenedorB, pero esta vez enlazándolo al contenedorA con la opción --link. Esto nos proporcionará un alias para acceder al contenedor enlazado. Realmente, Docker añadirá una entrada en el fichero /etc/hosts con el nombre del contenedor enlazada y el alias:

  $ docker stop contenedorB

  $ docker run -dit --rm --name contenedorB \ --link contenedorA:aliasA alpine sh

- **Enlazando contenedores por red:** Esta es la opción recomendada, pues el comando --link está en proceso de desaparecer. Para enlazar contenedores vía red vamos a crear una nueva red de tipo bridge y adherirlos a ella:

  $ docker network create --driver bridge miRed

  $ docker run -dit --rm --name contenedorA --network miRed alpine sh

  $ docker run -dit --rm --name contenedorB --network miRed alpine sh

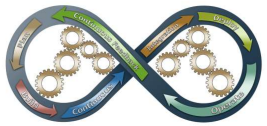
  $ docker network inspect miRed | jq .[0].Containers contenedorB - 172.18.0.3/16 contenedorA - 172.18.0.2/16

  $ docker attach contenedorA

  / # ping contenedorB

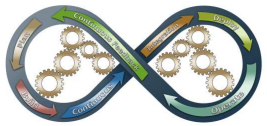
  PING contenedorB (172.18.0.3): 56 data bytes ...

  4 packets transmitted, 4 packets received, 0% packet loss



- La gran ventaja de esta opción es que escala con el número de contenedores y de redes. Por ejemplo, si creamos otra red más, podemos adherir un contenedor también a ella, creando de este modo una conexión entre redes distintas a nivel de IP:

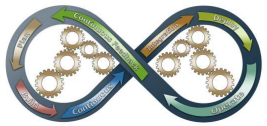
```
$ docker network create --driver bridge miRed2
$ docker network connect miRed2 contenedorB
 # Si inspeccionamos el contenedor lo veremos conectado a las dos redes
$ docker inspect contenedorB | jq .[0].Containers
miRed - 172.18.0.3
miRed2 - 172.19.0.2
```



- **Creación de una imagen:** Los contenedores de Docker están basados en imágenes previamente generadas que contienen todas las dependencias necesarias para nuestra aplicación o servicio. Aunque existen de manera pública multitud de imágenes listas para ser utilizadas, la mayoría de veces necesitaremos crear nuestras propias imágenes, habitualmente basadas en alguna ya disponible.

- Para automatizar la creación de imágenes de Docker describiremos los pasos necesarios para la construcción en un manifiesto de Docker llamado Dockerfile. Un fichero Dockerfile es un archivo de texto que contiene las instrucciones que se usarán para compilar y ejecutar una imagen de Docker.
  - En el Dockerfile se definen los siguientes aspectos de la imagen:
  - La imagen base sobre la que se creará la nueva imagen.
  - Comandos necesarios para actualizar el sistema operativo e instalar el software y dependencias necesarias.
  - El código o empaquetado de nuestra aplicación.
  - Puertos que expondrá el contenedor, así como la configuración de red y almacenamiento.
  - El comando que se ejecutará al iniciarse el contenedor.

```
$ docker build --help
```

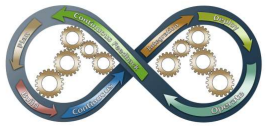


- **Creación de una imagen:** Vamos a generar una imagen de Docker para una aplicación sencilla de Python utilizando un Dockerfile. Supongamos que tenemos nuestra aplicación Python según la siguiente estructura de directorios:

```
app
├── Dockerfile
├── requirements.txt
└── src
    ├── ...
    └── server.py
```

Nuestro Dockerfile para nuestra aplicación podría tener un aspecto similar al siguiente:

```
# Establecemos la imagen base sobre la que construiremos nuestra imagen
FROM python:3.8.5

# Establecemos el directorio de trabajo del contenedor
WORKDIR /code

# Copiamos el fichero de dependencias al directorio de trabajo
COPY requirements.txt .

# Instalamos las dependencias definidas en el fichero
RUN pip install -r requirements.txt

# Copiamos nuestra aplicación de directorio local src al dir. de trabajo
COPY src/ .

# Configuramos los requerimientos de red
EXPOSE 5000

# Comando que se ejecutará al iniciarse el contenedor
CMD [ "python", "./server.py" ]
```



- **Creación de una imagen:** Una vez que tenemos creado nuestro fichero Dockerfile, podremos generar nuestra imagen con el comando docker build, indicando el nombre de la imagen y opcionalmente una etiqueta que, por defecto, será latest. Si nos fijamos en la salida del comando veremos que cada instrucción la ejecuta en un paso separado:

$ docker build -t python_app .

```
$ docker build -t python_app .
Sending build context to Docker daemon  5.12kB
Step 1/7 : FROM python:3.8.5
3.8.5: Pulling from library/python
...
Status: Downloaded newer image for python:3.8.5
 ---> 79cc46abd78d
Step 2/7 : WORKDIR /code
 ---> Running in a7cc669c40a4
Removing intermediate container a7cc669c40a4
 ---> 34a49c2912d5
Step 3/7 : COPY requirements.txt .
 ---> 0a1810e7a142
Step 4/7 : RUN pip install -r requirements.txt
 ---> Running in 4cad7d8b2c1f
...
---> 030d148118da
Step 5/7 : COPY src/ .
 ---> f82da4308992
Step 6/7 : EXPOSE 5000
 ---> Running in d47dbd078182
Removing intermediate container d47dbd078182
 ---> 965a5b1775dd
Step 7/7 : CMD [ "python", "./server.py" ]
 ---> Running in 4f1e1641d47b
Removing intermediate container 4f1e1641d47b
 ---> f855c94ad5df
Successfully built f855c94ad5df
Successfully tagged python_app:latest
```

- **Creación de una imagen:** Si hiciéramos cambios en el código fuente de la aplicación o en el fichero de dependencias podríamos regenerar la imagen volviendo a ejecutar el mismo comando docker build. Además, podemos ver las diferentes capas que componen una imagen utilizando el comando docker history:

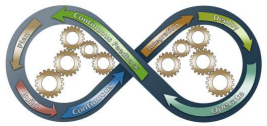
$ docker image history python_app:latest

```
$ docker image history python_app:latest
IMAGE          CREATED         CREATED BY                                      SIZE
859fb1db4a90   3 minutes ago   /bin/sh -c #(nop)  CMD ["python" "./server.p...  0B
867e1b296ff9   3 minutes ago   /bin/sh -c #(nop)  EXPOSE 5000:5000             0B
2cf8552ed9ca   3 minutes ago   /bin/sh -c #(nop) COPY dir:9b3e3bb266ad8957c...  164B
45dd0cfd49a7   3 minutes ago   /bin/sh -c pip install -r requirements.txt      9.63MB
9c7c03f19b8a   3 minutes ago   /bin/sh -c #(nop) COPY file:dcf08683799a1e65...  138
5a3811ed82ae   3 minutes ago   /bin/sh -c #(nop) WORKDIR /code                 0B
79cc46abd78d   5 days ago      /bin/sh -c #(nop)  CMD ["python3"]              0B
<missing>      5 days ago      /bin/sh -c set -ex;   wget -O get-pip.py "$P...  7.24MB
```

Ya solamente nos quedaría ejecutar un contenedor a partir de nuestra imagen, exponiendo el puerto del contenedor al host:

$ docker run -d -p 5000:5000 python_app

$ curl http://localhost:5000

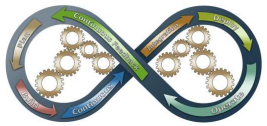


- **Creación de una imagen:** Argumentos y variables de entorno Es muy usual que necesitemos ciertos valores dentro del Dockerfile que no se conocen a la hora de escribirlo o que son credenciales privadas que, lógicamente, no deben estar en el fichero y se le deben inyectar en tiempo de creación de imagen. Para solucionar estos problemas tenemos dos opciones (complementarias, no excluyentes) que son los argumentos de entrada y las variables de entorno. Argumentos de entrada Se trata de parámetros que se pasan al Dockerfile a través de la sentencia de build:

```
$docker build --build-arg arg1 [--build-arg arg2 ...
```

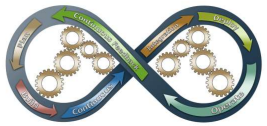
- Dentro del Dockerfile, estos parámetros se leen usando la palabra reservada ARGS, que admite un valor por defecto en caso de que el parámetro no se encuentre presente.

- Variables de entorno: El contenedor tiene acceso a todas las variables de entorno disponibles en el momento de ejecutar la construcción, pero solo puede usar aquellas que se declaren explícitamente con la palabra reservada ENV, que también admite un valor por defecto.



❑ Creación de una imagen:

```
FROM debian:buster-slim

ARG USER_NAME=tomcat
ARG WORKDIR="/home/$USER_NAME"
ARG USER_PROFILE_FILE="$WORKDIR/.bashrc"

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Elevamos permisos para crear un usuario
USER root
RUN set adduser --uid 1000 --disabled-password --system $USER_NAME

# Restringimos permisos para evitar crear recursos como root
USER $USER_NAME
ENV PATH=${WORKDIR}/.tomcat/bin:$PATH
```

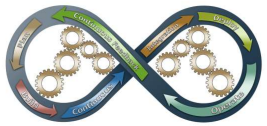


- **Docker Compose:** Docker Compose es una herramienta que nos permite simplificar el despliegue de aplicaciones multicontenedor como un único servicio, permitiéndonos gestionar fácilmente todo el ciclo de vida de los contenedores y otros objetos de la aplicación como volúmenes y redes. En lugar de crearnos nuestros propios scripts con llamadas al cliente de Docker para configurar y desplegar todos los objetos de nuestra aplicación (contenedores, redes, volúmenes, etc.).

- Docker Compose nos permite definir en ficheros YAML toda nuestra aplicación, su configuración y cómo se relacionan los objetos y sus dependencias.

- Será la propia herramienta Docker Compose la encargada de enviar las tareas necesarias a Docker Engine para crear los objetos y ejecutar los contenedores.

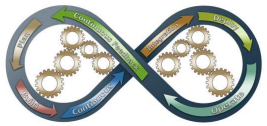
- Algunas de las ventajas de los servicios de Docker Compose son:
  - Permite el despliegue en el mismo host de múltiples entornos aislados de nuestra aplicación multicontenedor. Cada uno de estos entornos será un servicio en Docker Compose con un nombre de proyecto asociado.
  - Al iniciar un servicio de Docker Compose se buscan ejecuciones anteriores de los contenedores manteniendo sus volúmenes de datos, de manera que no perdamos información.
  - Al reiniciar un servicio de Docker Compose, se reutilizan aquellos contenedores cuya configuración no ha sido modificada, recreando únicamente aquellos que han sido modificados, acelerando así el reinicio.



- **Docker Compose:** Instalación de Docker Compose Antes de instalar Docker Compose deberemos tener instalado previamente Docker Engine. En los instaladores de Docker Desktop para Windows y Mac ya viene incluido Docker Compose, por lo que no será necesario hacer nada más. La instalación en Linux consistirá simplemente en descargar la herramienta y darle permisos de ejecución de la siguiente manera:

```
$ sudo curl -L \ "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

- Creación de servicios El uso de Docker Compose para la creación y despliegue de servicios se basa en los siguientes pasos:
  - En un directorio para el proyecto, crear el código y ficheros necesarios para nuestra aplicación, tal como haríamos habitualmente.
  - Crear los ficheros Dockerfile para las imágenes de nuestros contenedores en el directorio del proyecto.
  - Crear un fichero de definición de los servicios, volúmenes y redes en formato YAML llamado docker-compose.yml
  - Utilizar la herramienta docker-compose para crear los servicios y objetos Docker y gestionar su ciclo de vida



- **Docker Compose:** El formato del fichero Docker Compose Los ficheros de Docker Compose están escritos en formato YAML y, por defecto, tendrán el nombre de docker-compose.yml.
- En ellos definiremos los servicios, redes y volúmenes de nuestra aplicación. Existen varias versiones del formato de fichero de Docker Compose. Dependiendo de la versión del motor de Docker que estemos ejecutando soportaremos hasta una versión determinada.
- La última versión disponible de Docker Compose es la 3.8, soportada a partir de la versión 19.03 de Docker. Se recomienda comprobar que las opciones de configuración utilizadas en los ficheros YAML son soportadas por la versión de Docker que disponemos.
- Un fichero de Compose puede incluir las siguientes secciones a nivel raíz:
  - Version. Indica la versión de Compose utilizada en el fichero.
  - Services. Lista de servicios que componen nuestra aplicación.
  - Volumes. Permite crear volúmenes que serán utilizados por los servicios.
  - Networks. Permite crear volúmenes que serán utilizados por los servicios. En el siguiente ejemplo se muestra el contenido de un sencillo fichero de Compose en el que se definen dos servicios:

```
version: '3'
services:
 web:
  build: .
  ports:
   - "5000:5000"
 redis:
  image: "redis:alpine"
```

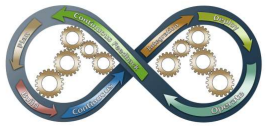
- **Docker Compose:** Definición de servicios en Compose Dentro de la key services se definirá cada uno de los servicios de los que consta nuestra aplicación multicontenedor, con una serie de parámetros adicionales. El nombre del servicio no es meramente informativo: se inyecta al resolver de cada contenedor de la misma red, por lo que un contenedor puede conectarse con otros usando su nombre, en lugar de tener que buscar su IP usando algún artificio. Esto es algo muy potente, pues permite simplificar enormemente la configuración de una aplicación: por ejemplo, podemos prefijar la URL de conexión a la base de datos de nuestra aplicación como "redis_database:8080" si el servicio de Redis se llama redis_database y expone el puerto 8080.

- La configuración build nos permite especificar la ruta del contexto de nuestro fichero Dockerfile: docs.docker.com

```
version: "3.8"
services:
 webapp:
  build: ./dir
```

- La configuración build soporta además otras configuraciones internamente. Por ejemplo, podemos especificar un nombre específico del Dockerfile:

```
build:
 context: ./dir
 dockerfile: Dockerfile.dev
```

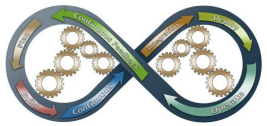


- **Docker Compose:**
  - Con la configuración image podemos indicar directamente la imagen que utilizará el contenedor, ya sea especificando su nombre o su identificador:

```
image: ubuntu
image: ubuntu:18.04
image: portainer/portainer-ce:alpine
```

  - Configuración de puertos La opción expose permite exponer puertos sin publicarlos en el host. Solamente serán accesibles por los servicios enlazados. Veamos un par de formas de definirlos:

```
# en yaml, ambas expresiones son equivalentes
expose: ["3000", “8000”]
expose:
  - "3000"
  - "8000"
```

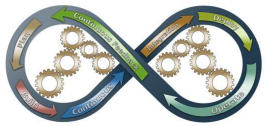


- **Docker Compose:** La opción ports permite publicar al host puertos del contenedor. Admite dos sintaxis, una corta especificando puerto_host:puerto_contenedor, y una larga con alguna opción adicional. Veamos algunos ejemplos:

```
ports:
  - "3000"
  - "8000:80"
  - "9090-9091:8080-8081"
  - "127.0.0.1:8001:8001"
  - "6060:6060/udp"

ports:
  - target: 80
    published: 8080
    protocol: tcp
    mode: host
```

- Ejecución de comandos: Las opciones command y entrypoint de Compose son equivalentes a las instrucciones CMD y ENTRYPOINT de los Dockerfiles que ya vimos. Recordemos que entrypoint sobrescribe a Command:

command: /app/entrypoint.sh
command: ["php", "-d", "vendor/bin/phpunit"]
entrypoint: /app/start-service.sh
entrypoint: [php, -d, vendor/bin/phpunit]



- **Docker Compose:** Variables de entorno Existen varias formas de pasar variables de entorno a los contenedores de los servicios. Podemos utilizar la configuración environment para indicar una lista de variables, o bien la configuración env_file para múltiples variables de entorno definidas en un fichero externo:

```
$ cat ./Docker/api/api.env
NODE_ENV=test

$ cat docker-compose.yml
version: '3'
services:
 api:
  image: 'node:6-alpine'
  env_file:
   - ./Docker/api/api.env
  environment:
   - NODE_ENV=production
```

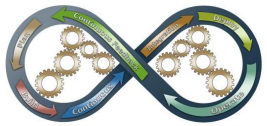
- En caso de tener varias definiciones de la misma variable de entorno, Compose utilizará el siguiente orden para elegir qué valor utilizar:
  - Fichero de Compose.
  - Variables de entorno del Shell.
  - Fichero de variables de entorno.
  - Variables definidas en el Dockerfile.
- La opción links permite enlazar otros servicios al contenedor permitiendo establecer un alias. Esta opción será eliminada en un futuro; en su lugar se recomienda utilizar la configuración de redes y establecer dependencias entre servicios:

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

- **Docker Compose:** Para establecer dependencias entre servicios utilizaremos la opción depends_on, permitiendo que un servicio espere a que otros estén arrancados antes de empezar su ejecución. Veamos un ejemplo:

```
services:
 web:
  build: .
  depends_on:
   - db
   - redis
 redis:
  image: redis
 db:
  image: postgres
```

- Montar volúmenes en un servicio La opción volumes, dentro de la definición de un servicio, nos permite montar rutas del host en el contenedor, o bien volúmenes a partir de su nombre. La sintaxis corta de esta opción nos permite utilizar el formato [ruta_host:]ruta_contenedor[:modo], mientras que la sintaxis larga permite configuraciones adicionales como, por ejemplo, el tipo de montaje.



- **Docker Compose:** Montar volúmenes en un servicio La opción volumes, dentro de la definición de un servicio, nos permite montar rutas del host en el contenedor, o bien volúmenes a partir de su nombre. La sintaxis corta de esta opción nos permite utilizar el formato [ruta_host:]ruta_contenedor[:modo], mientras que la sintaxis larga permite configuraciones adicionales como, por ejemplo, el tipo de montaje. Veamos algunos ejemplos

```
volumes:
  # Indicando solo la ruta del contenedor, se creará un nuevo volumen
  - /var/lib/mysql
  - /opt/data:/var/lib/mysql
  - ./cache:/tmp/cache
  - ~/configs:/etc/configs/:ro
  - datavolume:/var/lib/mysql

volumes:
  - type: volume
    source: mydata
    target: /data
    volume:
      nocopy: true

  - type: bind
    source: ./static
    target: /opt/app/static
```

- Definición de volúmenes en Compose Cuando referenciamos por nombre un volumen en la sección de los servicios, estos deben estar definidos en la sección volumes a nivel raíz del fichero de Compose. Esta sección nos permite la creación y definición de volúmenes asignando un nombre para poder ser referenciado en los servicios definidos.

```
services:
  db:
    image: postgres
    volumes:
      - datavolume:/var/lib/postgresql/data

volumes:
  datavolume:
    external: true
```

- Definición de redes en Compose Al igual que con los volúmenes, podemos definir a nivel raíz las redes que utilizan los servicios. Podemos crear nuevas redes o utilizar alguna ya existente, indicar el driver a utilizar y sus opciones, etc. Veamos un ejemplo de uso:

```
services:
  app:
    build: ./app
    networks:
      - frontend
      - backend
  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    name: frontend-network
  backend:
    external:
      name: existing-backend-network
```

- El comando docker-compose Utilizaremos la herramienta docker-compose para generar las imágenes y crear objetos definidos (volúmenes y redes) en el fichero de Compose. Por defecto usará el fichero con el nombre docker-compose.yaml del directorio actual, y como nombre del proyecto asignará el nombre del directorio en el que nos encontramos. Tanto el nombre del fichero como la ruta podemos modificarlos con los argumentos -f y -p respectivamente. Si nuestra aplicación es bastante compleja podríamos querer tener la definición de nuestra aplicación en varios ficheros YAML. Para ello usaremos tantas veces el parámetro -f como ficheros tengamos:

docker-compose build	Construye los servicios a partir de los Dockerfiles.
docker-compose up	Crea e inicia los contenedores del servicio.
docker-compose stop	Detiene un servicio, parando sus contenedores.
docker-compose start	Inicia un servicio parado previamente.
docker-compose down	Detiene y elimina un servicio, parando y eliminando los recursos asociados (contenedores, redes, imágenes, volúmenes).
docker-compose ps	Lista los contenedores de los servicios.
docker-compose exec	Ejecuta un comando en un contenedor del servicio.