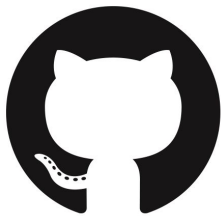


SEMINARIO

Introducción a los Contenedores Software





Hopla!
Software



formacion@hoplasoftware.com
team-docker@hoplasoftware.com

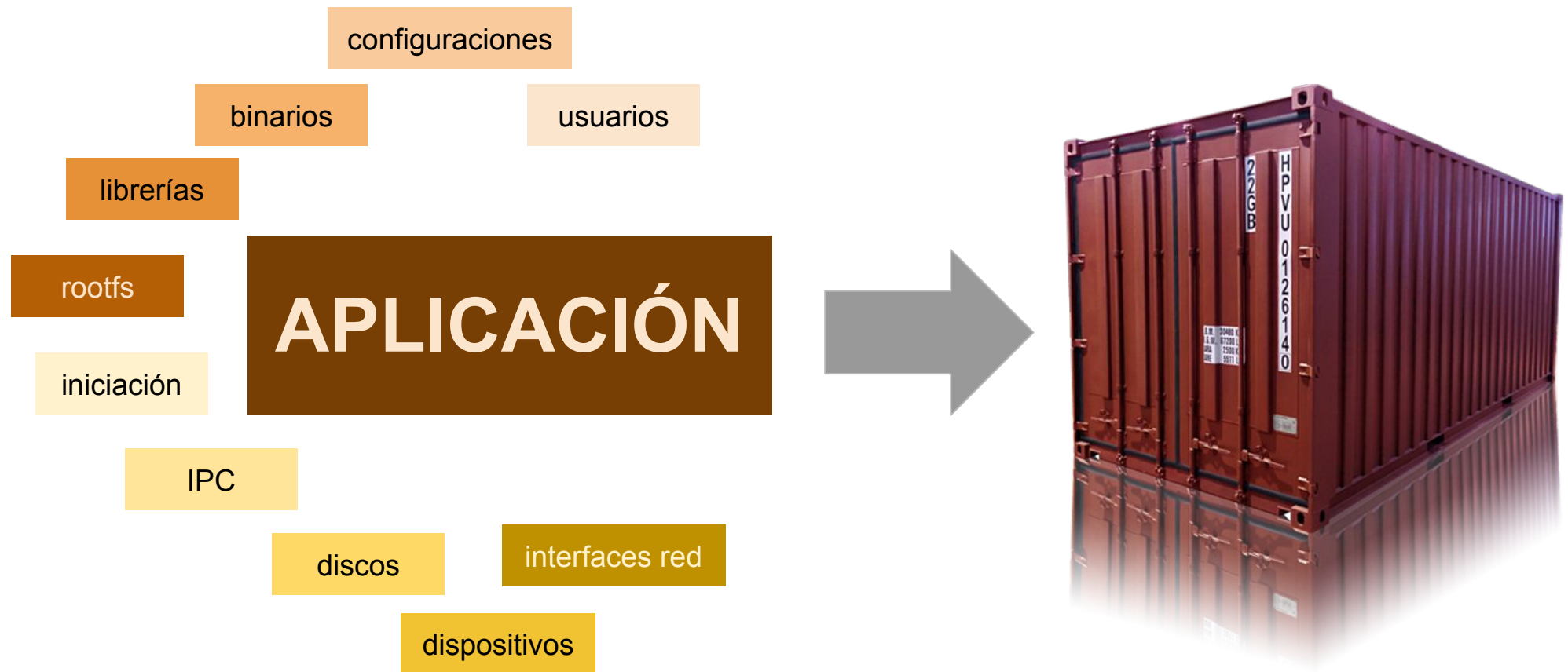
Introducción a los Contenedores Software

- **Qué son los Contenedores y Qué problemas resuelven?**
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Qué son los Contenedores Software?

Un contenedor software es la ejecución de procesos en un entorno conformado por un sistema de ficheros empaquetado con todo lo necesario para que una aplicación pueda ejecutarse.

Qué son los Contenedores Software?

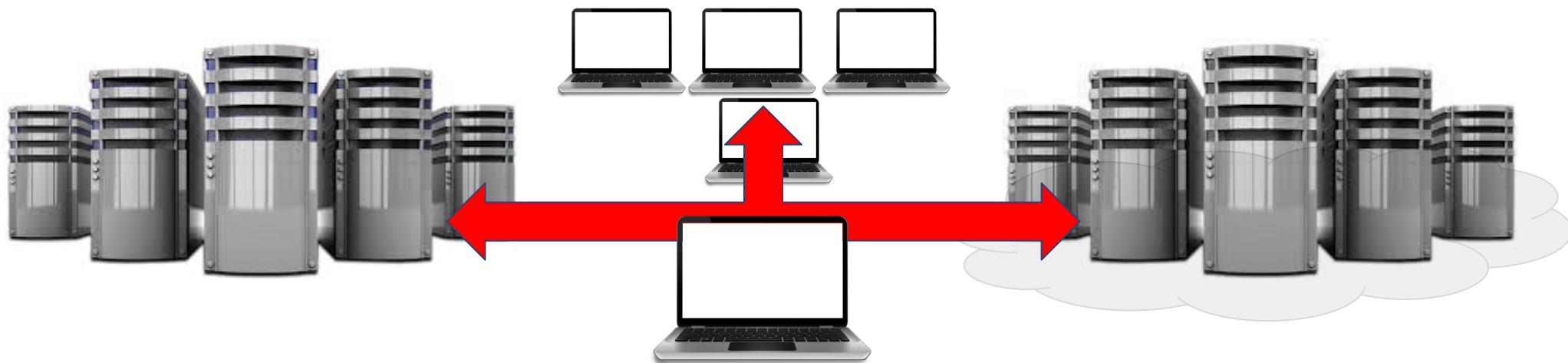


Qué problemas solucionan?

- En la actualidad disponemos de múltiples entornos para albergar nuestras aplicaciones e incluso podemos usar diferentes infraestructuras a la vez.
- Las aplicaciones deberían por tanto poder ejecutarse en **cualquier infraestructura en cualquier momento.**

Qué problemas se solucionan?

- Los contenedores nos proporcionan la capacidad de ejecutar nuestras aplicaciones en cualquier entorno de forma rápida y sin cambios en el aplicativo.



Qué problemas se solucionan?



Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- **Hardware vs Virtualización vs Contenedores**
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Aplicaciones asociadas a hardware – La Era del Metal

- Hasta hace unos años era habitual el uso de una infraestructura hardware completa dedicada a ejecutar un único aplicativo.



Aplicaciones asociadas a hardware – La Era del Metal

- Asociar nuestras aplicaciones directamente sobre hardware suponía:

- Provisión muy lenta y manual.
- Despliegue de aplicativos complejo.
- Escalado de aplicativo requiere añadir hardware.
- Alta disponibilidad requiere duplicar hardware.
- Migración y actualización de aplicativos complejas y lentas.
- Aplicaciones dependientes de hardware.
- Espacio físico en CPDs.
- Desaprovechamiento de recursos.

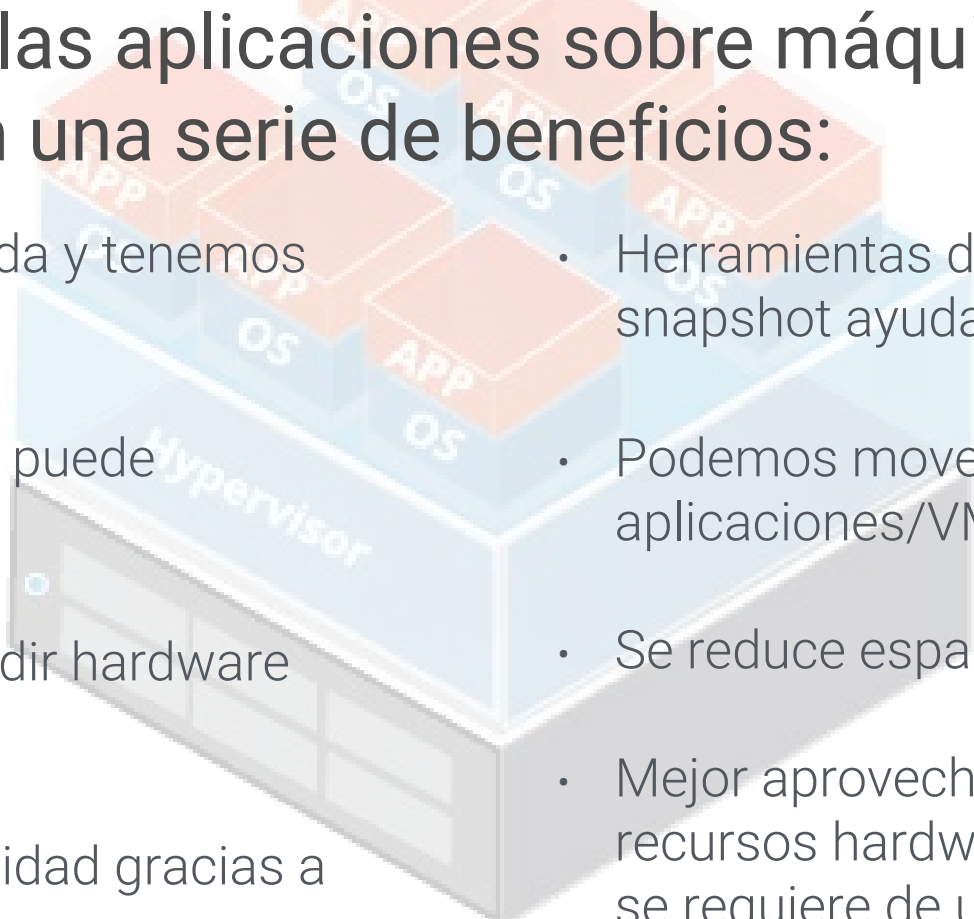
Aplicaciones sobre hardware virtualizado – La Era Virtual

- La aparición de la virtualización favoreció una primera optimización del hardware permitiendo el particionamiento de recursos en máquinas virtuales.



Aplicaciones sobre hardware virtualizado

– La Era Virtual

- 
- El diagrama ilustra la arquitectura de virtualización. En la base hay un bloque gris que representa el hardware. Encima de él se encuentra un bloque azul etiquetado como 'Hypervisor'. Sobre el Hypervisor se sitúan varias instancias de sistemas operativos virtuales, cada una etiquetada como 'OS'. Encima de cada 'OS' se ejecuta una aplicación, etiquetada como 'APP'. Las aplicaciones y sistemas operativos están representados como bloques de colores (naranja para APP y azul para OS) apilados en una estructura tridimensional.
- Al ejecutar las aplicaciones sobre máquinas virtuales, se obtienen una serie de beneficios:
 - Provisión rápida y tenemos APIs.
 - Herramientas de clonado y snapshot ayudan.
 - Despliegue se puede automatizar.
 - Podemos mover aplicaciones/VMs.
 - Podemos añadir hardware virtual.
 - Se reduce espacio en CPDs.
 - Alta disponibilidad gracias a hypervisor .
 - Mejor aprovechamiento de recursos hardware pero aún se requiere de un sistema operativo virtual.

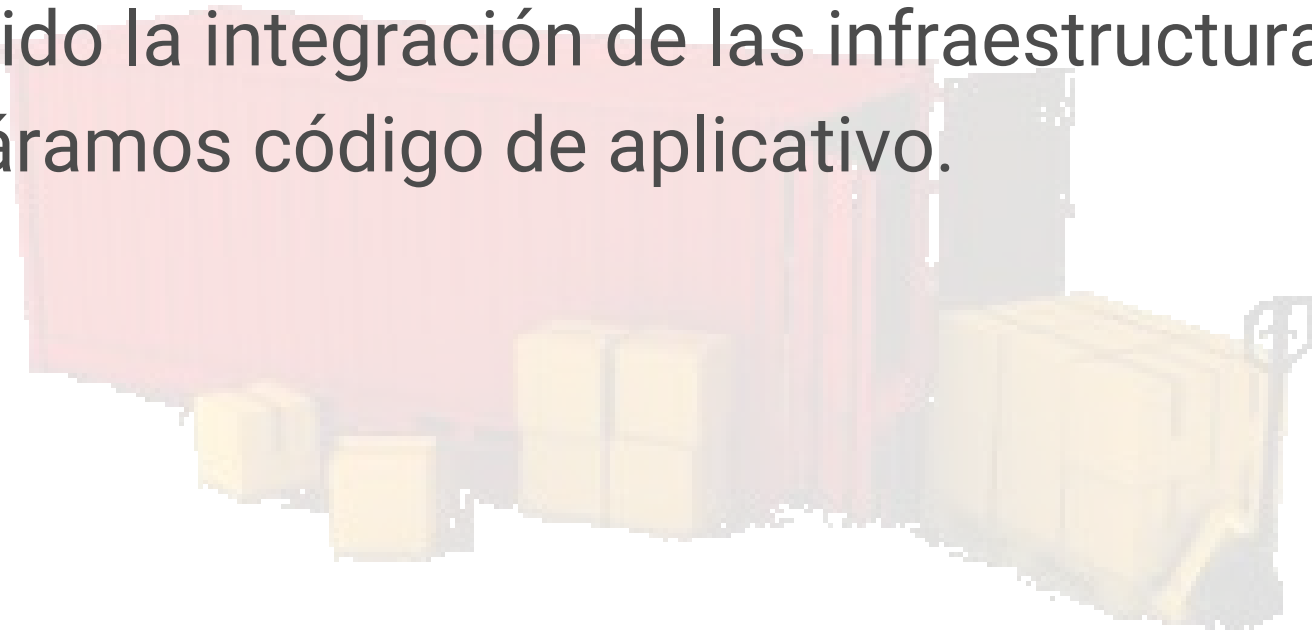
Aplicaciones en contenedores – La Era de las APIs y los contenedores

- La evolución del kernel de Linux fue añadiendo características mejoradas de gestión de recursos y aislamiento que nos han llevado a los contenedores software.



Aplicaciones en Contenedores – La Era de las APIs

- La evolución de los entornos SDN y SDS han permitido la integración de las infraestructuras como si tratáramos código de aplicativo.



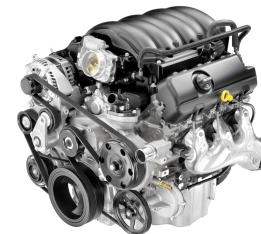
Aplicaciones en Contenedores – La Era de las APIs

• Beneficios:

- Los contenedores comparten el kernel, pero están aislados entre sí.
- Cada contenedor tiene su propio espacio de nombres para procesos, interrupciones, dispositivos, usuarios y comunicaciones.
- Recursos limitables y muy inferiores.
- Resiliencia e Inmutabilidad.
- No requieren de un sistema operativo completo, sólo ficheros de aplicativo y la estructura necesaria para su ejecución.
- Infraestructura reproducible..
- Gran portabilidad y despliegue en segundos.
- Provisión mediante código y versiones.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- **Conceptos Clave**
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito



Conceptos Clave - Instalaciones

Docker sobre Linux

De forma genérica, la instalación de la versión comunitaria de docker se instalará simplemente usando

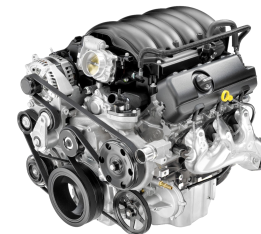
```
# curl -sSL https://get.docker.com/sh
```

Se puede instalar también usando los propios binarios y mediante paquetería (<https://docs.docker.com/engine/installation/>).

Docker sobre Windows

Docker sólo se puede instalar sobre Windows 2016 y Windows 10 Pro (con hyper-V). En todo caso, es posible instalar Docker ToolBox usando el siguiente enlace (<https://download.docker.com/win/stable/DockerToolbox.exe>).

Para realizar la instalación ejecutaremos como administrador <https://download.docker.com/win/stable/InstallDocker.msi> .



Conceptos Clave - Instalaciones

Docker ToolBox

- Docker ToolBox es la forma de ejecutar Docker sobre Windows en versiones no soportadas con Windows Containers (Windows 7, Windows 8 y Windows 10 Home).
- Proporciona un entorno de virtualización con Oracle VirtualBox sobre el que se provisionan máquinas virtuales, con docker-machine, preparadas para ejecutar contenedores.
- La configuración del cliente de docker con el engine remoto de la máquina virtual desplegada es transparente para el usuario.
- Es importante tener en cuenta las publicaciones de servicio.
- Incluye la herramienta gráfica Kitematic y una shell de ejecución Git.



Conceptos Clave - Instalaciones

Docker-Compose sobre Linux

De forma genérica, la instalación de docker-compose se realizará siguiendo el siguiente procedimiento (<https://docs.docker.com/compose/install/>)

```
# curl -L https://github.com/docker/compose/releases/download/1.13.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose  
# chmod 755 /usr/local/bin/docker-compose
```

Se puede instalar también usando los propios binarios y como módulo python (python pip) (pip install docker-compose).

Docker-Compose sobre Windows

Docker-Compose forma parte de Docker ToolBox por lo que no es necesario instalarlo en caso de usar esta opción.

En caso contrario, la instalación se realizará como administrador ejecutando la descarga del binario (<https://docs.docker.com/compose/install/>)

Invoke-WebRequest

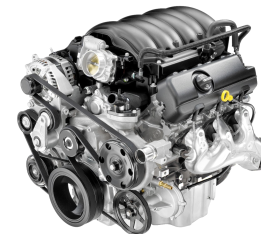
```
"https://github.com/docker/compose/releases/download/1.13.0/docker-compose-Windows-x86_64.exe"  
-UseBasicParsing -OutFile $Env:ProgramFiles\docker\docker-compose.exe
```



Conceptos Clave - Engine

Definición

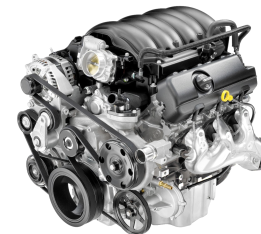
- El engine es el responsable de la ejecución de todas la gestión relacionada con los contenedores.
- Docker proporciona el software
 - **Servidor** - servicio-background/foreground
 - dockerd
 - runC
 - containerd
 - **Cliente (docker)** - requiere de servidor al que conectarse
- El motor/demonio/servidor de Docker es el encargado de todos los procesos relacionados con los contenedores e imágenes.
 - **Build** - Construcción de imágenes
 - **Ship** - Despliegue de imágenes
 - **Run** - Ejecución de contenedores



Conceptos Clave - Engine

Parametrización

- Es posible controlar el comportamiento del Engine modificando los parámetros de inicio del mismo en ***docker.service***.
- Podremos modificar:
 - Runtime de Docker
 - Parámetros de Comunicaciones de los Contenedores
 - Registros
 - Seguridad
 - Encriptación de las Comunicaciones
 - Gestión de Directivas
 - Mapeo de usuarios
 - Gestión de Logs
 - Drivers/Plugins

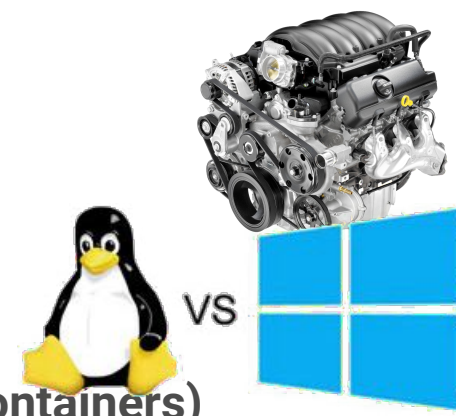


Conceptos Clave - Engine

Opciones

- Entre las diferentes opciones, destacamos las siguientes:
 - **--debug** → Modo Debug
 - **--graph <directorio_runtime_docker>** → Cambio del directorio de runtime (por defecto /var/lib/docker).
 - **--host <socket>** → Especificación de sockets para servicio (por defecto fd:// o unix:///var/run/docker.socket).
 - **--log-level <nivel>** → Nivel de log ("debug"|"info"|"warn"|"error"|"fatal", por defecto "info")
 - **--label <etiqueta>** → Especifica una etiqueta de localización para planificar contenedores.
 - **--storage-driver <driver>** → Permite especificar el driver a utilizar en runtime por los contenedores (devicemapper, aufs, overlay, btrfs).

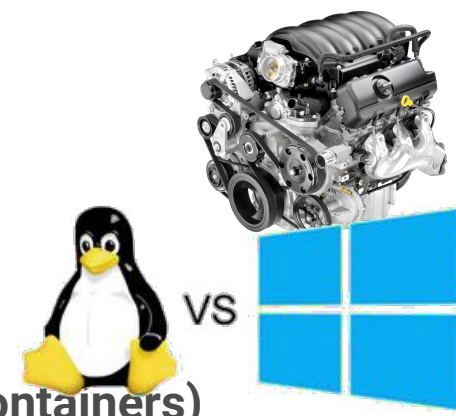
Conceptos Clave - Engine



Diferencias Engine Linux frente a Windows (Windows Containers)

- Similitudes:
 - Ambas soluciones se basan en la portabilidad de las aplicaciones.
 - Se ejecutan usando el kernel de sistema operativo.
 - No dependen de VM ni hipervisor, son nativos.
 - Se gestionan desde el cliente docker (también powershell en windows).
 - Únicamente pueden ejecutar aplicaciones nativas para el sistema operativo host.

Conceptos Clave - Engine



Diferencias Engine Linux frente a Windows (Windows Containers)

- Diferencias:
 - Solo se soportan unas versiones concretas de sistema operativo windows.
 - Las imágenes no son compatibles.
 - Las comunicaciones en windows son un poco diferentes (WinNAT por ejemplo o las redes overlay).
 - No son interoperables los cluster basados en diferente sistema de engine.
 - Windows sólo soporta Swarm como orquestador.



Conceptos Clave - Imágenes

Definición

- Las imágenes contienen un sistema de ficheros root con todo lo necesario para ejecutar nuestro aplicativo.
- No almacenan estado y su contenido es inmutable (plantillas).
- Se crean mediante definiciones en **Dockerfile**.
- Definición de imagen → HASH256:

HASH → ***repositorios:etiquetas***

```
sha256:c81ecce0cfbafb7d630098e8627dcdf3afb4a2f036518c5fe45e8d37c608dbc5 → varnish:alpine  
sha256:c81ecce0cfbafb7d630098e8627dcdf3afb4a2f036518c5fe45e8d37c608dbc5 → varnish:test1  
sha256:c81ecce0cfbafb7d630098e8627dcdf3afb4a2f036518c5fe45e8d37c608dbc5 → varnish:test2  
sha256:c81ecce0cfbafb7d630098e8627dcdf3afb4a2f036518c5fe45e8d37c608dbc5 → test1:varnish
```

- La etiqueta por defecto es “latest”.



Conceptos Clave - Imágenes

Uso

- Las imágenes se componen de una serie de capas (snapshots) de sistema de ficheros copy-on-write.

```
CMD nginx -g daemon off;
```

```
EXPOSE 80
```

```
COPY nginx_generic.conf /etc/nginx/nginx.conf
```

```
RUN ln -sf /dev/stdout /var/log/nginx/access.log \  
&& ln -sf /dev/stderr /var/log/nginx/error.log
```

```
RUN apk --no-cache --update add nginx curl
```

```
FROM alpine:3.5
```





Conceptos Clave - Imágenes

- Las imágenes se almacenan en registros.
 - DockerHub → **repositorio:[etiqueta]**
 - Registro Remoto →
registro_remoto:puerto/[usuario/grupo/]repositorio:[etiqueta]
 - Además, dependiendo del registro, las imágenes podrán ser:
 - Públicas
 - Privadas
 - Oficiales
 - Certificadas
- } **DockerHub**



Conceptos Clave - Imágenes

Dockerfile

- Los ficheros de construcción de imágenes se denominan Dockerfile y Docker tiene una serie de primitivas disponibles para la creación de las imágenes.
- Cada línea de instrucciones en Dockerfile ejecutada en la creación de una imagen da lugar a una capa nueva.

FROM

RUN

COPY / ADD

WORKDIR

EXPOSE

CMD

ENTRYPOINT



Conceptos Clave - Imágenes

Acciones

- Acciones disponibles:
 - Construcción/Borrado de imágenes.
 - build
 - rm
 - Descarga/Subida usando registros remotos.
 - pull
 - push
 - Listado y revisión de información.
 - ls
 - inspect
 - Cambios en el etiquetado y en la propia imagen.
 - tag
 - commit



Conceptos Clave - Imágenes

Construcción:

```
$ docker image build -t nombre_imagen[:etiqueta] [-f Fichero_Construccion] contexto
```

Laboratorio 1.

Construcción de una imagen a partir de un fichero de construcción Dockerfile sencillo.

1. Crear los directorios LABS y LABS1.

```
$ mkdir -p LABS/LABS1
```

```
$ cd LABS/LAB1
```

2. Descargar el fichero Dockerfile y nginx.conf

```
$ curl -sSL
```

```
https://raw.githubusercontent.com/hopla-training/hands-on-labs/master/mynginx/Dockerfile -o Dockerfile
```

```
$ curl -sSL
```

```
https://raw.githubusercontent.com/hopla-training/hands-on-labs/master/mynginx/mynginx.conf -o mynginx.conf
```



Conceptos Clave - Imágenes

Construcción:

```
$ docker image build -t nombre_imagen[:etiqueta] [-f Fichero_Construccion] contexto
```

Laboratorio 1 (Continuación).

Construcción de una imagen a partir de un fichero de construcción Dockerfile sencillo.

- 3. Construimos la imagen llamada "mynginx:1.0" desde el directorio LABS/LAB1**

```
$ docker image build -t mynginx:1.0 .
```

- 4. Listamos las imágenes existentes en nuestro sistema.**

```
$ docker image ls
```




Conceptos Clave - Imágenes

Etiquetado:

```
$ docker image tag repo_orig:etiq_orig repo_dest:etiq_dest
```

Laboratorio 2.

Cuando ejecutamos ***docker image tag***, creamos una nueva nomenclatura para un mismo ID de imagen.

1. Creamos una nueva etiqueta usando como origen la creada anteriormente (mynginx:1.0).

```
$ docker image tag mynginx:1.0 mynginx:2.0
```



Conceptos Clave - Imágenes

Borrado:

```
$ docker image rm <repositorio:etiqueta>
```

Laboratorio 3.

Borrar una imagen supone realmente el borrado de sus pares **repositorio:etiqueta**.

1. Listamos las imágenes actuales

```
$ docker image ls
```

2. Borramos la imagen etiquetada anteriormente y observamos la salida

```
$ docker image rm mynginx:2.0
```

No se borrará el identificador de la imagen hasta que no se hayan borrado todas las referencias repositorio:etiqueta referentes al mismo; por tanto, la imagen con ID dado, seguirá existiendo en nuestro sistema.



Conceptos Clave - Imágenes

Revisión de configuración de imagen:

```
$ docker image inspect <repositorio:etiqueta>
```

Laboratorio 4.

Podemos consultar la metainformación de una imagen usando ***docker image inspect repositorio:etiqueta***.

1. Listamos las imágenes actuales

```
$ docker image ls
```

2. Revisamos la información

```
$ docker image inspect mynginx:2.0
```

La salida del comando nos mostrará un objeto Json al que podremos filtrar usando `--format` para especificar la salida que queremos obtener. Por ejemplo:

```
$ docker image inspect --format '{{.Config.ExposedPorts}}' mynginx:2.0  
map[80/tcp:{}]
```



Conceptos Clave - Imágenes

Descarga y Subida de imágenes:

```
$ docker image pull <repositorio:etiqueta>
```

```
$ docker image push <repositorio:etiqueta>
```

Laboratorio 5.

Para este laboratorio será necesario contar con una cuenta en DockerHub (<https://hub.docker.com>).

Descargamos imágenes usando ***docker image pull repositorio:etiqueta***.

1. Descargamos una imagen de busybox

```
$ docker image pull busybox
```

2. Creamos un etiquetado nuevo haciendo referencia a un repositorio de usuario (cada alumno usará su usuario de DockerHub previamente creado):

```
$ docker image tag busybox:latest nombre_de_usuario/busybox:custom
```

3. Subiremos ahora la imagen recién etiquetada:

```
$ docker image push nombre_de_usuario/busybox:custom
```

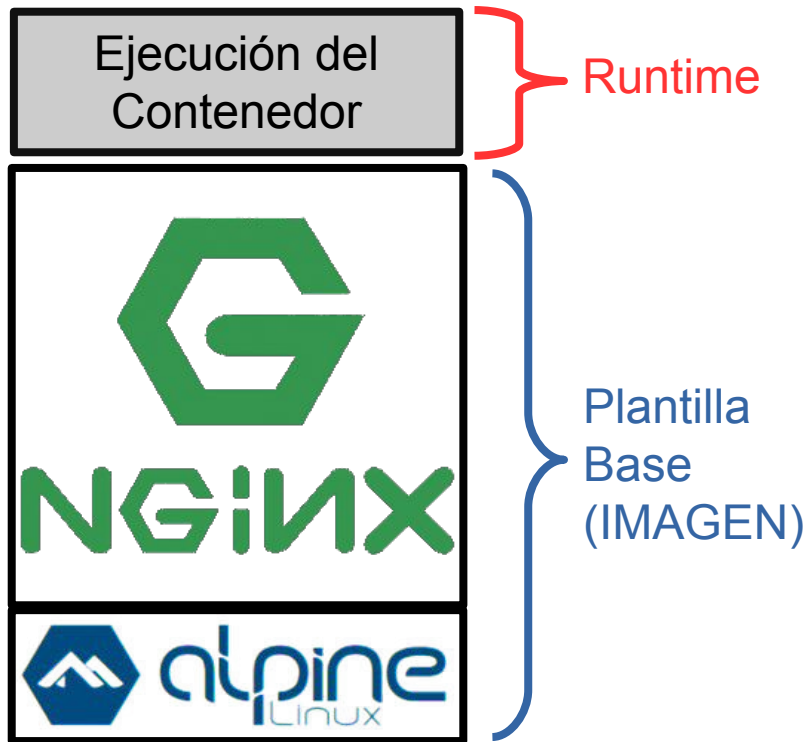
Conceptos Clave - Contenedores



Definiciones

- Runtime de ejecución creado a partir de una plantilla.
- Únicamente almacena cambios en disco.
- Un proceso principal que gestiona el estado.
- Comparten kernel del host y se gestiona los recursos con cgroups.
- Cada contenedor tiene su propio espacio:
 - Procesos.
 - Red.
 - Usuarios.
 - IPC.
 - Sistema de Ficheros.
 - Dispositivos.
- Aislamiento total entre contenedores.

Conceptos Clave - Contenedores



Acciones:

- Creación y/o Ejecución.
- Parada/Arranque/Reinicio.
- Vincular/Desvincular.
- Ejecutar comandos dentro de un contenedor.
- Matar y Destruir contenedores.
- Revisar logs.
- Ver configuración y actualizar recursos.

Conceptos Clave - Contenedores



Creación/Ejecución:

```
$ docker run [--name nombre] [-t] [-i] [-d] [-P] \  
[-p puerto_host:puerto_contenedor] imagen[:etiqueta] [línea de comandos/argumentos]
```

Laboratorio 6.

Creación de contenedores interactivos.

1. **Creación de un contenedor interactivo llamado mishell con la imagen mynginx:1.0 con una shell 'sh'.**
\$ docker run --name mishell -ti mynginx:1.0 sh
2. **Observamos qué ocurre cuando salimos de la shell ejecutada en el contenedor.**
3. **Creamos ahora un contenedor interactivo llamado mishell2 con la imagen mynginx:1.0 con una shell 'sh'**
\$ docker run -d --name mishell2 -ti mynginx:1.0 sh
4. **Observamos que nos devuelve prompt.**

Conceptos Clave - Contenedores



Creación/Ejecución:

```
$ docker run [--name nombre] [-t] [-i] [-d] [-P] \  
[-p puerto_host:puerto_contenedor] imagen[:etiqueta] [línea de comandos/argumentos]
```

Laboratorio 6 (Continuación).

Creación de contenedores interactivos.

5. Listamos los contenedores en ejecución.

```
$ docker ps
```

6. Listamos todos los contenedores (incluyendo los parados).

```
$ docker ps -a
```


Conceptos Clave - Contenedores



Parada y Arranque de Contenedores:

\$ docker stop/start/restart

Laboratorio 7.

Operaciones de parada y arranque del contenedor mishell.

- 1. Listamos todos los contenedores (incluyendo los parados).**
\$ docker ps -a
- 2. Observamos la salida y tomando el identificador del contenedor 'mishell', lo iniciamos de nuevo vinculándonos a él.**
\$ docker start -a -i <ID_CONTENEDOR_MISHELL>
- 3. Podemos desvincularnos del contenedor usando Ctrl+P-Q sin que esto finalice la shell.**
- 4. Observamos que nos devuelve prompt en nuestro host.**

Conceptos Clave - Contenedores



Parada y Arranque de Contenedores:

\$ docker stop/start/restart

Laboratorio 7 (Continuación).

Operaciones de parada y arranque del contenedor mishell.

5. Listamos los contenedores en ejecución.

\$ docker ps

6. Paramos el contenedor 'mishell'

\$ docker stop mishell

7. Ejecutamos un reinicio de 'mishell' para que lo levante de nuevo (no se para en este caso y no nos vinculamos a él).

\$ docker restart mishell

Conceptos Clave - Contenedores



Parada y Arranque de Contenedores:

\$ docker stop/start/restart

Laboratorio 7 (Continuación).

Operaciones de parada y arranque del contenedor mishell.

La parada de un contenedor se realiza con `docker stop`, que envía una señal `SIGTERM` sobre el proceso principal. Si no se finaliza en un tiempo (configurable y por defecto de 10 segundos), se envía `SIGKILL`. También es posible directamente ejecutar `docker kill` para enviar `SIGKILL`.

Conceptos Clave - Contenedores



Conversión a imagen y Borrado de Contenedores:

\$ docker rm / commit

Laboratorio 8.

Operaciones de conversión (snapshot) a imagen y borrado de contenedores.

1. **Listamos todos los contenedores (incluyendo los parados).**

\$ docker ps -a

2. **Creamos una imagen nueva partiendo del contenedor 'mishell' llamada mynginx:2.0**

\$ docker commit mishell mynginx:2.0

Esta imagen tiene cambios respecto a mynginx:1.0 en los ficheros de histórico de la shell por ejemplo.

3. **Tratamos de borrar ahora el contenedor 'mishell2' y observamos el error.**

\$ docker rm mishell2

Podemos borrar contenedores parados únicamente o usar --force.

\$ docker rm --force mishell2

Conceptos Clave - Contenedores



Revisión de Logs y Ejecución de comandos en Contenedores:

\$ docker logs / docker exec

Laboratorio 9.

En esta práctica crearemos un contenedor con un servicio web con la imagen nginx:alpine en el puerto 8080, observaremos su log y ejecutaremos una shell dentro del mismo.

1. Creamos un contenedor llamado 'mishell'.

\$ docker run --name mishell -d -p 8080:80 nginx:alpine

2. Observamos el error debido a que ya existe un contenedor llamado 'mishell'

3. Creamos entonces un contenedor llamado 'webserver'

\$ docker run --name webserver -d -p 8080:80 nginx:alpine

La imagen base nginx:alpine redirige los log de accesos y errores a la salida estándar y de error; por lo que esto es lo que veremos usando docker logs.

Conceptos Clave - Contenedores



Revisión de Logs y Ejecución de comandos en Contenedores:

\$ docker logs / docker exec

Laboratorio 9 (Continuación).

En esta práctica crearemos un contenedor con un servicio web con la imagen nginx:alpine en el puerto 8080, observaremos su log y ejecutaremos una shell dentro del mismo.

4. Desde un navegador podremos conectarnos a <http://localhost:8080> (puede diferir en caso de usar AWS/VMs o provisión mediante docker-machine).
5. Observamos la salida de docker logs y vemos las peticiones del navegador.
6. Por último, lanzamos una shell interactiva sobre el contenedor en ejecución 'webserver'
\$ docker exec -ti webserver sh

Conceptos Clave - Contenedores



Publicación de puertos de procesos:

```
$ docker run --publish / --publish-all
```

Laboratorio 10.

En esta práctica crearemos un contenedor con un servicio web en el puerto 8081 y otro dejando que docker seleccione un puerto aleatoriamente.

1. Creamos un contenedor llamado 'web1' usando el puerto de host 8081 asociado al 80 del contenedor.
\$ docker run --name web1 -d -p 8081:80 nginx:alpine
2. Creamos ahora otro contenedor llamado 'web2' sin especificar esta vez el puerto de host y permitiendo a docker que seleccione uno disponible (usará el rango 32768 a 65000).

```
$ docker run --name web1 -d -P 80 nginx:alpine
```

Si no especificamos ningún puerto, el demonio usará los definidos en la imagen base para el contenedor.

Conceptos Clave - Contenedores



Publicación de puertos de procesos:

```
$ docker run --publish / --publish-all
```

Laboratorio 10 (Continuación).

En esta práctica crearemos un contenedor con un servicio web en el puerto 8081 y otro dejando que docker seleccione un puerto aleatoriamente.

3. Desde un navegador o usando curl, podremos conectarnos a <http://localhost:8081> (puede diferir en caso de usar AWS/VMs o provisión mediante docker-machine).
4. Usando `docker ps` o `docker port <contenedor>` podremos ver los puertos asignados al contenedor 'web2' y posteriormente verificaremos el acceso usando curl o un navegador sobre `http://localhost:<PUERTO>` (puede diferir en caso de usar AWS/VMs o provisión mediante docker-machine).

Conceptos Clave - Contenedores



Recursos (CPU/Memoria/IO/Red):

```
$ docker run --memory / --cpus / --device-write-iops
```

Podremos especificar recursos de Memoria, CPU y Entrada/Salida de disco, y además es posible configurar su entorno de red.

Algunos de los recursos pueden modificarse en tiempo de ejecución, incluso conectarse a redes existentes del entorno.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- **Herramientas**
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Herramientas



Docker

- El binario docker comunica directamente con el API expuesto por el servidor y Swarm.
- Las comunicaciones con el servidor no están encriptadas por defecto.
- Es el encargado de indicar al servidor las tareas a realizar, tanto en el demonio local como en swarm:
 - Empaquetado/Construcción de imágenes
 - Distribución de imágenes
 - Creación y Ejecución de contenedores

Herramientas



Docker-Compose

- Sirve para gestionar de forma unificada una agrupación de contenedores.
- Se comporta igual que el cliente docker.
- En la versión actual únicamente permite ejecuciones en un nodo.

Herramientas



Docker-Machine

- Provisión de nodos preparados para ejecutar contenedores.
- Existen diferentes drivers para proveedores en la nube (AWS, Azure, Digital Ocean), en infraestructura local (Openstack, VMWare, Xen, SSH), hardware (HP Oneview) e infraestructuras hiperconvergentes (Nutanix).
- La provisión despliega Boot2Docker.
 - Sistema operativo mínimo basado en Tiny Core Linux, ejecutado completamente en memoria (~38MB).
 - Persistencia de sistema de ficheros de docker.

Herramientas



Plugins

- Extienden las funcionalidades de Docker:
 - Volúmenes
 - Comunicaciones
 - Autorización
- Desde la versión 17.03 existen los plugins certificados, cuya integración con la infraestructura está garantizada por el fabricante.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- **Seguridad**
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito



Seguridad

Seguridad en engine

- La primera medida de seguridad siempre es el acceso al socket `/var/run/docker.socket`.
- Los contenedores basan su seguridad en el uso de namespaces y control groups, de manera que los procesos ejecutados dentro disponen de una serie de recursos encapsulados y aislados.
- Es posible el uso de user namespaces con mapeos explícitos de usuarios para evitar accesos root inesperados.



Seguridad

Seguridad en engine

- Existe integración con Iptables para permitir que Docker gestione las reglas del mismo.
- Es imprescindible el uso de encriptación TLS en las comunicaciones entre los clientes y los hosts y entre los propios hosts en un cluster.
- Además, existe integración con las herramientas propias de gestión de seguridad de sistema como SELinux, AppArmor o GRSEC que añaden aún más aislamiento y protección.



Seguridad

Seguridad en contenedores

- Los contenedores son más seguros que un sistema completo porque la superficie de ataque muy inferior.
- Los contenedores nacen con el concepto de aislamiento total en mente.
- Podemos desplegar contenedores privilegiados o especificar las capacidades que el kernel facilitará a los mismos (para un listado completo léase [man capabilities](#)).
- Contenedores de sólo lectura (--read-only).



Seguridad

Uso de TLS

- Usaremos TLS para encriptar las comunicaciones entre los hosts y entre los clientes y ellos.
- TLS se basa en el uso de claves de los distintos componentes, certificadas y firmados por una entidad certificadora.
- Para encriptar la comunicación necesitaremos:
 - Crear una CA o usar la nuestra.
 - Crear una clave de servidor y firmarla con la CA.
 - Crear una clave de cliente y firmarla con la CA.



Seguridad

Uso de TLS

- Procedimiento:

- Creamos CA y su certificado:

```
$ openssl genrsa -aes256 -out ca-key.pem 2048
```

```
$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
```

- Creamos una clave de servidor, su petición de firma y la firmamos con la CA:

```
$ openssl genrsa -out server-key.pem 2048
```

```
$ openssl req -subj "/CN=<host>" -new -key server-key.pem -out server.csr
```

```
$ echo subjectAltName = IP:<IP_servidor>,IP:127.0.0.1 >extfile.cnf
```

```
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem \  
-CAcreateserial -out server-cert.pem -extfile extfile.cnf
```



Seguridad

Uso de TLS

- Procedimiento (Continuación):

- Creamos una clave de servidor, su petición de firma y la firmamos con la CA:

```
$ openssl genrsa -out client-key.pem 2048
```

```
$ openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr
```

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

```
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \  
-CAcreateserial -out client-cert.pem -extfile extfile.cnf
```

- Habilitamos el demonio con TLS:

```
--tlsverify
```

```
--tlscacert=<path_ca.pem>
```

```
--tlscert=<path_server-cert.pem>
```

```
--tlskey=<path_server-key.pem>
```

```
-H=0.0.0.0:2376
```

Es necesario preparar los certificados creados en un directorio de servidor y de cliente de forma separada, con los permisos adecuados (0400 para ca-key.pem, client-key.pem y server-key.pem y 0444 para ca.pem, server-cert.pem y client-cert.pem)



Seguridad

Uso de TLS

- Habilitamos el cliente con TLS:

```
--tlsverify  
--tlscacert=<path_ca.pem>  
--tlscert=<client-cert.pem>  
--tlskey=<client-key.pem>  
-H=<servidor>:2376
```

Podemos usar directamente cert.pem y key.pem para poder especificar directamente el path completo a los certificados:

```
DOCKER_TLS_VERIFY=1  
DOCKER_CERTS_PATH=<path_a_certificados>  
DOCKER_HOST=<servidor:puerto>
```



Seguridad

Seguridad en Imágenes

- Se debe buscar el mínimo tamaño de componentes para minimizar la superficie de ataque de los contenedores.
- Es imprescindible construir las imágenes indicando el usuario de ejecución del proceso maestro del contenedor resultado.
- Usar el nuevo objeto “secret” para almacenar configuraciones y credenciales (sólo con swarm).
- El firmado de imágenes asegurará la procedencia y veracidad.
- Docker Datacenter (DTR) incluye escaneo de imágenes en busca de vulnerabilidades.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- **Almacenamiento de Datos**
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Almacenamiento - Volúmenes



Definiciones

- Un volumen es un directorio del contenedor designado para almacenar datos persistentes.
- Los volúmenes existen de forma independiente de los contenedores que los usan.
- Podremos usar los volúmenes para intercambiar datos entre contenedores e incluso con el propio host anfitrión.

Almacenamiento - Volúmenes



Definiciones

- Los volúmenes sirven para saltarse el sistema copy-on-write, ya que su contenido se escribe directamente.
- Podemos extender la funcionalidad de los volúmenes mediante plugins.
- Diferenciaremos entre:
 - Volúmenes definidos en imagen
 - Volúmenes nominales
 - Directorios/Ficheros de Host como volúmenes

Almacenamiento - Volúmenes



Acciones:

- Creación / Borrado.
 - Listar volúmenes.
 - Revisar Contenedores que usan el volumen y su configuración.
-
- Cuando creamos volúmenes en SWARM, el ámbito deja de ser local de manera que la metainformación necesaria para “contactar” con el mismo está disponible en el cluster.
 - Cuando usamos plugins para crear volúmenes, podemos disponer de opciones adicionales, por ejemplo indicando el tamaño del mismo a provisionar en el sistema de almacenamiento asociado.
 - Cuando usamos imágenes con volúmenes definidos en su creación, se creará el volumen indicado en la definición (se podría obtener usando *inspect* sobre la *imagen*) pero sin nombre asociado.
 - Para montar directorios del hosts, usaremos el path completo.

Almacenamiento - Volúmenes



Creación, Listado y Borrado de Volúmenes:

\$ docker volume create / docker volume rm /docker rm -v <container>

Laboratorio 10.

Crearemos un volumen llamado 'mydata' y haremos uso de él en un contenedor busybox.

1. Creamos un volumen llamado 'mydata' y listamos los volúmenes existentes:

```
$ docker volume create --name mydata  
$ docker volume ls
```

2. Creamos ahora un contenedor con busybox y usamos el volumen en /tmp para poner un fichero llamado 'test1':

```
$ docker run --name uno -ti --rm busybox -v mydata:/tmp sh  
> # touch /tmp/test1
```

Como hemos usado el argumento '--rm' en la creación, el contenedor se borra al salir de la shell

Almacenamiento - Volúmenes



Creación, Listado y Borrado de Volúmenes:

\$ docker volume create / docker volume rm /docker rm -v <container>

Laboratorio 10 (Continuación).

Crearemos un volumen llamado 'mydata' y haremos uso de él en un contenedor busybox.

- 3. Creamos ahora otro contenedor usando el mismo volumen anterior y observamos que el fichero sigue en su sitio:**

```
$ docker run --name dos -ti --rm busybox -v mydata:/tmp sh  
> # ls -lart /tmp/test1
```

- 4. Para finalizar, borramos el volumen y listamos de nuevo:**

```
$ docker volume rm mydata  
$ docker volume ls
```

Almacenamiento - Volúmenes



Inspeccionar un volumen para ver su configuración:

```
$ docker volume inspect
```

Laboratorio 11.

Crearemos un volumen llamado 'mydata' y observaremos sus propiedades.

1. Creamos un volumen llamado 'mydata' y listamos los volúmenes existentes:

```
$ docker volume create --name mydata
```

```
$ docker volume ls
```

2. Revisamos ahora la configuración del volumen anterior:

```
$ docker volume inspect mydata
```

Almacenamiento - Volúmenes



Montar un directorio de host dentro de un Contenedor:

```
$ docker run -v <HOST_PATH>:<CONTAINER_PATH>
```

Laboratorio 12.

Montaremos el directorio /tmp dentro de un contenedor y crearemos un fichero.

1. Creamos un contenedor montando /tmp de host dentro:

```
$ docker run -ti --rm -v /tmp:/tmp busybox touch /tmp/test_desde_contenedor
```

2. Observamos el fichero creado recientemente. Qué podemos ver acerca del propietario y sus permisos?

```
$ ls -l /tmp/test_desde_contenedor
```

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- **Comunicaciones**
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Comunicaciones



Definiciones

- Las comunicaciones en los contenedores se basan fundamentalmente en interfaces virtuales asociados a interfaces de bridge creados en el host.
- Por defecto, Docker creará el interface de bridge docker0 durante su instalación.
- Es posible especificar el direccionamiento a usar en la creación de los contenedores tanto a nivel global (demonio) como en cada uno de ellos.
- Cada red desplegada en Docker cuenta con su propia resolución DNS.

Comunicaciones



Definiciones

- Sin plugins adicionales, Docker permite las siguientes especificaciones de red, cada una de ellas con su uso más apropiado:
 - none
 - bridge (por defecto)
 - host
 - overlay
 - macvlan

Comunicaciones



Definiciones

- Cada vez que se crea una red bridge, se crea un interface bridge asociado sobre el que se vincularán los interfaces de los contenedores.
- Los contenedores asociados a diferentes redes no se ven entre sí.
- Las redes overlay se extienden en todos los nodos de un cluster swarm y los contenedores asociados a ellas podrán comunicarse entre ellos independientemente del host en el que se encuentre.

Comunicaciones



Definiciones

- Las aplicaciones arrancadas dentro de un contenedor no escucharán al exterior salvo que se publique explícitamente el puerto asociado.
- Salvo que se especifique lo contrario, Docker gestiona de forma transparente iptables, modificando sus reglas siempre que sea necesario, añadiendo o quitando tanto accesos como routing y NAT hacia los contenedores.

Comunicaciones



Uso de interface por defecto docker0:

```
$ docker run --net=bridge
```

Laboratorio 13.

Crearemos un contenedor usando el interface por defecto y veremos su ip asignada.

1. Creamos un contenedor usando la red por defecto con una shell interactiva:

```
$ docker run -ti --rm --name lab13 -busybox sh
```

2. Observamos ahora sus ips. Qué direccionamiento tiene?.

```
> # ip address
```

3. Revisamos su hostname y luego salimos de la shell.

```
> # hostname
```

4. Observamos ahora la configuración de la red 'bridge'

```
$ docker network inspect bridge
```

Comunicaciones



Uso de interface nulo:

```
$ docker run --net=none
```

Laboratorio 14.

Crearemos un contenedor usando el interface nulo y veremos su configuración de red.

1. Creamos un contenedor usando la red 'none' con una shell interactiva:

```
$ docker run -ti --rm --net=none --name lab14 -busybox sh
```

2. Observamos ahora su ip y luego salimos de la shell.

```
> # ip address
```

3. Qué interfaces tiene?.

Comunicaciones



Uso de red de host anfitrión:

```
$ docker run --net=host
```

Laboratorio 15.

Crearemos un contenedor usando el interface nulo y veremos su configuración de red.

1. Creamos un contenedor usando la red 'none' con una shell interactiva:

```
$ docker run -ti --rm --net=host --name lab15 -busybox sh
```

2. Observamos ahora su ip.

```
> # ip address
```

3. Qué interfaces tiene?.

4. Revisamos su hostname y luego salimos de la shell.

```
> # hostname
```

5. Qué observamos en la configuración de red del contenedor?

Comunicaciones



Uso de red bridge :

```
$ docker network create -d bridge --subnet W.X.Y.Z/máscara <NET_NAME>
```

Laboratorio 16.

Crearemos una red de bridge nueva y un contenedor que use esta red al que luego vincularemos con la red 'bridge' original.

1. Creamos una red de bridge llamada 'mired' y listamos las redes disponibles:

```
$ docker network create -d bridge --subnet 10.10.10.10/24 mired  
$ docker network ls
```

2. Ejecutamos un contenedor y observamos su ip (recordemos que podemos usar Ctrl+P+Q para desvincularnos).

```
$ docker run -ti --rm --net=host --name lab16 -busybox sh  
> # ip address
```

3. Qué interfaces tiene?.

Comunicaciones



Uso de red bridge :

```
$ docker network create -d bridge --subnet W.X.Y.Z/máscara <NET_NAME>
```

Laboratorio 16 (Continuación).

Crearemos una red de bridge nueva y un contenedor que use esta red al que luego vincularemos con la red 'bridge' original.

4. Desde el terminal del host, conectaremos el contenedor 'lab16' con la red de bridge por defecto llamada 'bridge'.

```
$ docker network connect bridge lab16
```

5. Revisamos de nuevo las ips usando 'exec' sobre el contenedor 'lab16' y ejecutando 'ip address'.

```
$ docker exec lab16 ip address
```

Comunicaciones



Redes macvlan:

Las redes de tipo macvlan (usando el driver macVlan) proporcionan un acceso directo al interface de red físico del host anfitrión. Esto mejora el acceso de los contenedores asociados a la red y permite el uso de IPs reales de nuestro entorno.

Como contrapunto, cabe indicar que no es posible gestionarlas como objetos del cluster debido a su ámbito local.

Comunicaciones



Uso de redes overlay :

Por su parte, las redes overlay se definen explícitamente en entornos clusterizados y se extienden en todos los hosts que forman parte del mismo. Esto permite desplegar aplicaciones en el cluster con componentes distribuidos en diferentes hosts.

Por otra parte, el orquestador Swarm hace uso de las redes overlay también para distribuir de la forma más eficiente las distintas tareas de un servicio replicado, permitiendo el balanceo interno entre ellos (VIP o DNSRR) en la red llamada 'ingress'.

Veremos ejemplos de este tipo de redes cuando estudiemos Swarm (Orquestación).

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones

→ Orquestación

- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Orquestación



SWARM

- Swarm es el orquestador oficial de Docker.
- Basado en Swarmkit:
 - Raft - Consensus
 - KV distribuido
 - Servicios - Tareas
 - Basado en estado (resiliencia)
- No requiere software adicional y se integra con todo el ecosistema.

Orquestación



SWARM

- Raft:
 - Requiere un número impar de “managers” para asegurar la elección de un único líder.
 - Alta disponibilidad mínima 3 “managers”.
 - Únicamente el líder realiza cambios en KV.
 - Sólo los managers gestionan el cluster.

Orquestación



SWARM

- Principales características:
 - Gestión automatizada (Roles - Managers/Leader - Workers)
 - Objetos de cluster en KV distribuido y configurables dinámicamente.
 - Cluster Swarm
 - Nodos
 - Servicios -> Tareas -> Contenedores
 - Secretos
 - Redes Overlay
 - Redes transversales entre los nodos (Overlay) incluidas (encriptables).
 - Planificación de Tareas siguiendo el principio de resiliencia.
 - Encriptación TLS de comunicaciones entre nodos del cluster out-of-box.
 - Routing Mesh proporcionará una forma sencilla de acceder a los servicios.

Orquestación



SWARM

- Al crear un cluster:
 - Se elige un líder para gestionar los cambios.
 - Se crea una base KV de datos distribuida.
 - Se crea una entidad certificadora y se firman las claves del primer nodo para así poder cifrar las comunicaciones con TLS.
 - Se crea una red interna overlay llamada “ingres” que se usará para publicar los puertos expuestos de los servicios.

Orquestación



Creación de un cluster:

```
$ docker swarm init [--listen-addr ip|interface]
```

Laboratorio 18.

Crearemos un cluster nuevo y añadimos 3 nodos worker.

1. Creamos un cluster nuevo en node1:

```
$ docker swarm init
```

Nos aparece las indicaciones para dar de alta los nuevos hosts worker dentro del cluster.

Si perdemos el token, podremos volver a recuperarlo (para cada rol) usando *docker swarm join-token* *<rol_a_consultar>*

2. Nos conectamos a los nodos de uno en uno y ejecutamos la siguiente línea (copia de las instrucciones obtenidas al crear el cluster)

```
(node2)$ docker swarm join \  
--token TOKEN_OBTENIDO_EN_EJECUCIÓN_ANTERIOR \  
IP_DEL_NODO_LEADER_DEL_CLUSTER:2377
```

Orquestación



Creación de un cluster:

```
$ docker swarm init [--listen-addr ip|interface]
```

Laboratorio 18 (Continuación).

Crearemos un cluster nuevo y añadimos 3 nodos worker.

- 3. Por último, desde el nodo principal (leader), revisamos el cluster y observamos los estados y roles de los nodos.**

```
$ docker node ls
```

Podremos cambiar los roles de los nodos del cluster siempre que sea necesario; por ejemplo ante la caída de uno de los managers, podremos promover un worker para mantener el consenso.

Orquestación



Gestión de Roles:

```
$ docker node promote/demote <node>
```

Laboratorio 19.

Con 3 nodos worker y un único manager no aseguramos la disponibilidad del cluster. Realizaremos los cambios necesarios para tener 3 managers y un único worker.

1. **Desde el único manager (además líder), ejecutamos el cambio de rol del nodo node2 de worker a manager:**

```
$ docker node promote node2
```

2. **Realizamos el mismo cambio con node3:**

```
$ docker node promote node3
```

3. **Revisamos los cambios desde cualquiera de los nodos manager (node1, node2 o node3).**

```
$ docker node ls
```

Orquestación



SERVICIOS y TAREAS

- En un cluster Swarm únicamente lanzaremos servicios. Cualquier contenedor ejecutado en sus nodos queda fuera del alcance y gestión del cluster.
- Un servicio es la unidad de despliegue de procesos en el cluster.
- Cada servicio se define por su estado y el número de tareas necesarias para su despliegue.
- Las tareas son la unidad de ejecución en los nodos componentes del cluster, y ejecutan exactamente un contenedor.
- “Rolling Update” permite actualización sin pérdida de servicio.

Orquestación



SERVICIOS y TAREAS

- Por defecto, un servicio que publique un puerto, balanceará las peticiones recibidas entre las distintas tareas, asociando una IP virtual internamente en la red interna de publicación del cluster.
- Es posible especificar el balanceo entre tareas mediante DNSRR.
- La publicación de puertos de servicio se realiza en todos los nodos del cluster.
- Los puertos de publicación dinámicamente comenzarán en 30000, en lugar de 32768.

Orquestación



Creación y publicación de un servicio:

```
$ docker service create [--name nombre] [--mode global|replicated] \  
  [--replicas numero] [--publish <puerto_host>:<puerto_contenedor>] \  
<imagen[:etiqueta]> [linea_comandos]
```

Laboratorio 20.

Crearemos un servicio llamado 'webserver', publicado en el puerto 8080 de todos los hosts del cluster.

1. Creamos un servicio nuevo en node1 con la imagen nginx:alpine:

```
$ docker service create --name webserver --publish 8080:80 nginx:alpine
```

2. Usamos docker service ls para verificar que el servicio se ha desplegado y debe tener una réplica arrancada:

```
$ docker service ls
```

Orquestación



Creación y publicación de un servicio:

```
$ docker service create [--name nombre] [--mode global|replicated] \  
  [--replicas numero] [--publish <puerto_host>:<puerto_contenedor>] \  
  <imagen[:etiqueta]> [línea_comandos]
```

Laboratorio 20 (Continuación).

Crearemos un servicio llamado 'webserver', publicado en el puerto 8080 de todos los hosts del cluster.

3. Verificamos que el servicio está disponible en local usando curl o un navegador sobre <http://localhost:8080>.
4. Además, routing mesh publica el puerto en todos los hosts del cluster, de manera que podremos acceder a http://ip_node2:8080, http://ip_node3:8080, y lo mismo con node4.

Orquestación



Verificación de la distribución de tareas y escalado:

```
$ docker service ps <nombre_servicio>
```

Laboratorio 21.

Verificaremos la distribución de tareas del servicio llamado 'webserver', aumentaremos el número de réplicas y observaremos de nuevo el resultado.

1. **Para revisar las tareas creadas asociadas al servicio 'webserver', usamos docker service ps:**
\$ docker service ps webserver
2. **Aumentamos ahora el número de réplicas usando docker service scale <servicio>=<número> o bien docker service update --replicas <número> <servicio>**
\$ docker service scale webserver=5

Orquestación



Verificación de la distribución de tareas y escalado:

```
$ docker service ps <nombre_servicio>
```

Laboratorio 21 (Continuación).

Verificaremos la distribución de tareas del servicio llamado 'webserver', aumentaremos el número de réplicas y observaremos de nuevo el resultado.

3. **Verificamos el despliegue usado `docker service ps`, observando los hosts del cluster que ahora ejecutan tareas:**

```
$ docker service ps webserver
```

4. **Qué ocurre si matamos un contenedor en uno de los hosts?**

Orquestación



Mantenimiento de nodos de cluster (Vaciado y Pausa):

```
$ docker node update --availability [drain/pause] <nodo>
```

Laboratorio 22.

En este laboratorio vaciaremos de tareas uno de los nodos y pondremos en pause otro de ellos, observando lo que ocurre cuando aumentamos de nuevo el número de tareas del servicio (escalado).

1. **Para vaciar de tareas del nodo node1, configuraremos su disponibilidad en el cluster como “drain”:**

```
$ docker node update --availability drain node1
```

```
$ docker node ls
```

2. **Observamos el cambio en el nodo con “docker node ls” y luego qué ocurre con las tareas que se ejecutan en él.**

Orquestación



Mantenimiento de nodos de cluster (Vaciado y Pausa):

```
$ docker node update --availability [drain/pause] <nodo>
```

Laboratorio 22 (Continuación).

En este laboratorio vaciaremos de tareas uno de los nodos y pondremos en pause otro de ellos, observando lo que ocurre cuando aumentamos de nuevo el número de tareas del servicio (escalado).

3. Pasemos ahora el nodo node2 a pausa.

```
$ docker node update --availability pause node2
```

```
$ docker node ls
```

4. Ahora escalamos de nuevo el servicio y observamos qué ocurre con la distribución de tareas en los hosts.

```
$ docker service scale webserver=10
```

```
$ docker service ps webserver
```

Orquestación



Servicios Globales y Replicados:

```
$ docker service create [--mode global|replicated]
```

Laboratorio 23.

Hasta ahora hemos estado creando servicios replicados (con tareas distribuidas en el cluster). Los servicios globales crearán únicamente una tarea asociada, pero el orquestador planificará esta tarea en todos y cada uno de los nodos del cluster.

1. Creamos un servicio global llamado “global-webserver” en el puerto 9090 usando de nuevo nginx:alpine:

```
$ docker service create --mode global --publish 9090:80 nginx:alpine  
$ docker service ls  
$ docker service ps global-webserver
```

2. Observamos la distribución del servicio.

El orquestador no replanificará nunca tareas tras caídas de host, vaciado o pausa, salvo en los casos en los que los servicios sean globales o se fuerce explícitamente la replanificación usando *docker service update --force <servicio>*

Orquestación



Orquestación de Resiliencia de Servicios

- Swarm proporciona recuperación ante fallos out-of-box:
 - Ante caída de un contenedor → Se crea una nueva tarea y por ello un nuevo contenedor para recuperar la situación de salud.
 - Ante caída de host → Se replanifican las tareas en ejecución en el host en los otros nodos, creando nuevas tareas y sus contenedores asociados para recuperar la situación de salud de todos los servicios afectados.
- Es conveniente desplegar servicios con más de una réplica para mejorar la gestión ante errores, pero debe soportar las condiciones de vip o dnsrr.

Orquestación



Orquestación de Resiliencia de Servicios:

Laboratorio 24.

Este laboratorio mostrará la recuperación de las tareas desplegadas de un servicio en caso de caída de un host (parada de dockerd).

1. **Partimos de la situación en la que todos los nodos se encuentran activos (son susceptibles de ejecutar tareas). Verificamos esta situación con node ls:**

```
$ docker node ls
```

2. **Seleccionamos uno de los nodos que actualmente ejecuta tareas (podemos usar docker node ps) y paramos el demonio de docker.**

```
$ docker node ps node1
```

```
(nodo seleccionado) # systemctl stop docker (en sistemas con SystemD)
```

o bien

```
(nodo seleccionado) # /etc/init.d/docker stop (si usamos Boot2Docker)
```

3. **Observamos la redistribución de tareas en los diferentes nodos del cluster..**

Orquestación



Redes en cluster

- Los servicios desplegados en un cluster, únicamente usarán redes overlay (actualmente pueden desplegarse redes de host también).
- Las redes overlay comunican transversalmente todas las tareas de servicios desplegados sobre ella.
- Cuando creamos una red overlay, creamos un objeto nuevo de cluster, visible en los managers y en todos los host que ejecutan una tarea de un servicio que use esta red.

Orquestación



Redes en cluster

- Cuando se crea el cluster, se crea la red ingres, dedicada a gestión del cluster y publicación de servicios replicados (VIP y DNSRR).
- Se asocia un DNS interno a cada red overlay con los servicios desplegados sobre ella.
- Las redes overlay son redes VxLan gestionadas automáticamente por el orquestador y con capacidad para ser encriptadas.

Orquestación



Redes en cluster

- Los elementos desplegados en una red overlay están aislados de los elementos de otras redes del mismo cluster, salvo que expongamos los procesos de los contenedores.

Orquestación



Redes en cluster:

```
$ docker network create -d overlay <nombre_red>
```

Laboratorio 25.

En este laboratorio crearemos una red transversal en el cluster llamada 'mired' y crearemos dos servicios vinculados a la misma.

1. Creamos una red llamada 'mired':

```
$ docker network create -d overlay mired
```

2. Creamos ahora un servicio llamado 'simplestdb' basado en `hopla/simplest-lab:simplestdb` (no publicaremos ningún puerto).

```
$ docker service create --network=mired \  
  --name simplestdb hopla/simplest-lab:simplestdb
```

3. Creamos ahora un servicio llamado 'simplestapp' basado en `hopla/simplest-lab:simplestapp`.

```
$ docker service create --network=mired --publish 8080:3000 \  
  --name simplestapp hopla/simplest-lab:simplestapp
```

Orquestación



Redes en cluster:

```
$ docker network create -d overlay <nombre_red>
```

Laboratorio 25 (continuación).

Revisamos ahora la aplicación en el puerto 8080 (que se corresponde con el 3000 de la aplicación desplegada en el servicio).

4. Nos conectaremos a la aplicación mediante curl o un navegador en <http://localhost:8080> (routing mesh nos llevará al contenedor correcto).
5. Podemos revisar la image `hopla/simplest-lab:simplestdb` y observaremos que se trata de un servicio **postgreSQL** que se publicará internamente en 5432, pero que queda confinado en la red 'mired'.

```
$ docker image inspect hopla/simplest-lab:simplestdb
```

Orquestación



Redes en cluster:

```
$ docker network create -d overlay <nombre_red>
```

Laboratorio 25 (continuación).

Únicamente los servicios desplegados dentro de la misma red 'mired' podrán consumir las exposiciones internas de procesos (en este ejemplo, el puerto 5432 sólo es accesible por el servicio del aplicativo, en la misma red).

En la aplicación desplegada en <http://localhost:8080>, podremos observar también el comportamiento del balanceo interno con VIP usando réplicas.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- **Despliegue de Servicios y Aplicaciones**
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Despliegue de Servicios y Aplicaciones



Despliegue de Aplicaciones

- Definiremos una aplicación como un conjunto de componentes que trabajan entre ellos para ofrecer un servicio.
- Docker permite el despliegue de aplicaciones usando:
 - Compose
 - Stacks

Despliegue de Servicios y Aplicaciones



Compose

- Compose define una aplicación directamente basada en contenedores (multi-contenedores).
- Los servicios definidos mediante compose no son servicios de cluster, son componentes de aplicativo.
- En la versión actual, docker-compose únicamente despliega aplicaciones en un host, no permite su uso en cluster.
- Cada aplicación crea y usa una red independiente aislada del resto de aplicaciones y contenedores del host.

Despliegue de Servicios y Aplicaciones



Compose

- Con el comando docker-compose podremos:
 - Construir imágenes → build
 - Distribuir imágenes → pull/push
 - Ejecutar los contenedores definidos → up/down/start/stop
 - Gestionar los despliegues → ps/logs

Despliegue de Servicios y Aplicaciones



Stack (application bundle)

- La evolución de swarm originó una nueva forma de desplegar objetos de forma unificada en el cluster mediante “stacks”.
- Ahora sí, un stack es un conjunto de servicios que trabajan conjuntamente.
- El despliegue de un stack crea una red overlay aislada para las comunicaciones de todos sus servicios.
- Docker stack no tiene acciones para construir ni gestionar imágenes; por lo que las imágenes deben encontrarse en registro.

Despliegue de Servicios y Aplicaciones



Ficheros Compose

- Tanto el despliegue de aplicaciones con docker-compose como con stack (application bundle) se definirán usando ficheros docker-compose.
- Los ficheros compose tienen formato yaml y disponen de primitivas similares a las utilizadas en la ejecución manual de servicios y contenedores.
- Los despliegues con docker-compose y con docker stack requieren de un nombre identificativo de aplicativo (por defecto será el del propio directorio de despliegue).

Despliegue de Servicios y Aplicaciones



Ficheros Compose

- Usaremos mínimo la versión 2 de sintaxis, pero si usamos docker stack será necesaria la versión 3.
- Definiremos:
 - Servicios
 - Volúmenes
 - Redes
 - Secretos

```
version: "2"
services:
  simplestdb:
    build: simplestdb
    image: hopla/simplest-lab:simplestdb
    volumes:
      - DBDATA:/PGDATA
  simplestapp:
    build: simplestapp
    image: hopla/simplest-lab:simplestapp
    depends_on:
      - simplestdb
    ports:
      - 8080:3000
volumes:
  DBDATA:
```

Despliegue de Servicios y Aplicaciones



Ficheros Compose

- build

La directiva “build” requiere del contexto (directorio) y creará una imagen etiquetada como “proyecto_nombre-servicio:latest”, donde el proyecto se pasará como argumento o usará el propio nombre de directorio. En caso de que se implemente junto con la directiva “image”, usará su nomenclatura en la construcción.

Despliegue de Servicios y Aplicaciones



Ficheros Compose

- image

La directiva “image” por su parte define la imagen base para la creación de los contenedores. Podremos usar su etiquetado o su hash sha256. Si existe en local se ejecutará el contenedor y si no es así, se descargará de la forma habitual del registro indicado.

Despliegue de Servicios y Aplicaciones



Despliegue con docker-compose:

```
$ docker-compose [-p proyecto] [-f fichero_compose] up [-d]
```

Laboratorio 26.

En esta práctica, desplegamos una sencilla aplicación con un componente que actuará como frontend que realizará un sencillo balanceo sobre los backend desarrollados en nodejs que guardarán los hits de acceso a ellos mismos en una base de datos postgresql (<https://github.com/hopla-training/simplest-lab>).

1. Preparamos un directorio de trabajo y descargamos el fichero mentorsday-lab.yml.

```
$ curl -sSL
```

```
https://raw.githubusercontent.com/hopla-training/simplest-lab/master/mentorsday-lab.yml \  
-o simplestlab.yml
```

2. Observamos los diferentes componentes del despliegue y verificamos la sintaxis usando docker-compose config

```
$ docker-compose -f simplestlab.yml -p composelab config
```

Despliegue de Servicios y Aplicaciones



Despliegue con docker-compose:

```
$ docker-compose [-p proyecto] [-f fichero_compose] up [-d]
```

Laboratorio 26 (Continuación).

3. **Descargamos las imágenes necesarias usando docker-compose pull (este paso no es necesario para el despliegue realmente).**
\$ docker-compose -f simplestlab.yml -p composelab pull
4. **Desplegamos el aplicativo en background usando docker-compose up -d.**
\$ docker-compose -f simplestlab.yml -p composelab up -d
5. **Observamos el despliegue del aplicativo y comprobamos el estado y los puertos usando docker-compose ps**
\$ docker-compose -f simplestlab.yml -p composelab ps
6. **Nos conectamos con nuestro navegador al aplicativo en el puerto <http://localhost:8080>.**

Despliegue de Servicios y Aplicaciones



Despliegue con docker-compose:

```
$ docker-compose [-p proyecto] [-f fichero_compose] up [-d]
```

Laboratorio 26 (Continuación).

- 7. Vemos los diferentes componentes del aplicativo y observamos el log usando docker-compose logs.**

```
$ docker-compose -f simplestlab.yml -p composelab logs
```

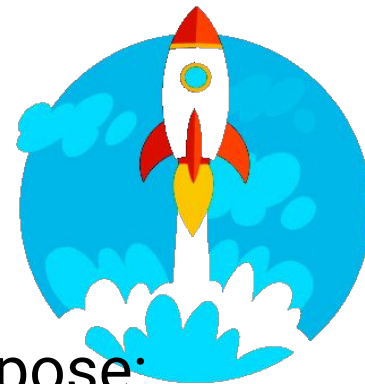
- 8. Como podemos comprobar, no todos los componentes exponen sus procesos al exterior.**

```
$ docker-compose -f simplestlab.yml -p composelab logs
```

- 9. Revisamos además las redes y volúmenes creados para el aplicativo.**

```
$ docker network ls && docker volume ls
```


Despliegue de Servicios y Aplicaciones



Gestión de componentes desplegados con docker-compose:

```
$ docker-compose [-p proyecto] [-f fichero_compose] ps | scale
```

Laboratorio 27.

Partiendo del despliegue anterior, escalaremos el componente backend para ver el efecto que se obtiene.

1. Observamos los servicios desplegados.

```
$ docker-compose -f simplestlab.yml -p composelab config --services  
$ docker-compose -f simplestlab.yml -p composelab ps
```

2. El servicio 'app' es el backend de aplicativo, escalamos este componente a 3 instancias por ejemplo.

```
$ docker-compose -f simplestlab.yml -p composelab scale app=3
```

3. Observamos de nuevo el despliegue y nos conectamos al frontal en <http://localhost:8080> para ver qué ha sucedido (refrescar varias veces y observar las ips de los componentes y la gráfica con los hits).

```
$ docker-compose -f simplestlab.yml -p composelab ps
```

Despliegue de Servicios y Aplicaciones



Eliminación de despliegues con docker-compose:

```
$ docker-compose [-p proyecto] [-f fichero_compose] down/stop/start
```

Laboratorio 28.

Partiendo del despliegue anterior, pararemos el servicio frontal, lo arrancaremos y limpiaremos completamente el despliegue.

1. **Paramos es componente app usando docker-compose stop y observamos el resultado con docker-compose ps.**

```
$ docker-compose -f simplestlab.yml -p composelab stop app  
$ docker-compose -f simplestlab.yml -p composelab ps
```

2. **Arrancamos de nuevo el componente 'app' usando docker-compose start.**

```
$ docker-compose -f simplestlab.yml -p composelab start
```

3. **Para finalizar con docker-compose, borramos completamente el despliegue con docker-compose down.**

```
$ docker-compose -f simplestlab.yml -p composelab down
```

Despliegue de Servicios y Aplicaciones



Despliegues con stack:

```
$ docker stack deploy
```

Laboratorio 29.

En esta práctica, desplegamos una sencilla aplicación con un componente que actuará como frontend que realizará un sencillo balanceo sobre los backend desarrollados en nodejs que guardarán los hits de acceso a ellos mismos en una base de datos postgresql (<https://github.com/hopla-training/simplest-lab>). La diferencia entre stack y docker-compose es que stack nos permite desplegar en el swarm.

1. En esta ocasión, usaremos el fichero `simplest-lab.V3.yml` para realizar el despliegue en swarm.

Descargamos el fichero en un manager de swarm.

```
$ curl -sSL
```

```
https://raw.githubusercontent.com/hopla-training/simplest-lab/master/simplest-lab.V3.yml \  
-o simplestlab.v3.yml
```

2. Desplegamos el aplicativo usando `docker stack deploy`.

```
$ docker stack deploy -c simplestlab.v3.yml stacklab
```

Despliegue de Servicios y Aplicaciones



Despliegues con stack:

```
$ docker stack deploy [-c fichero_compose] nombre_stack
```

Laboratorio 29 (Continuación).

En caso de usar repositorios protegidos (privados), será necesario siempre usar el parámetro `--with-registry-auth`.

1. **Observamos el despliegue del aplicativo en el cluster usando docker stack ps.**
\$ docker stack ps stacklab
2. **Nos conectamos con nuestro navegador al aplicativo en el puerto <http://localhost:8080>.**
3. **Para observar los servicios desplegados, usaremos docker stack services, que nos indicará además los puertos escuchando.**
\$ docker stack services stacklab
4. **Revisamos además las redes y volúmenes creados para el despliegue.**
\$ docker network ls && docker volume ls

Despliegue de Servicios y Aplicaciones



Gestión de componentes desplegados con stack:

\$ docker stack ps | docker service scale

Laboratorio 30.

Partiendo del despliegue anterior, escalaremos el componente backend para ver el efecto que se obtiene.

1. **Observamos los servicios desplegados.**

\$ docker stack services stacklab

2. **El servicio 'stacklab_app' es el backend de aplicativo, escalamos este componente a 3 instancias por ejemplo.**

\$ docker service scale stacklab_app=3

3. **Observamos de nuevo el despliegue y nos conectamos al frontal en <http://localhost:8080> para ver qué ha sucedido (refrescar varias veces y observar las ips de los componentes y la gráfica con los hits).**

\$ docker stack ps stacklab

Despliegue de Servicios y Aplicaciones



Eliminación de despliegues realizados con stack:

```
$ docker stack rm
```

Laboratorio 31.

Partiendo del despliegue anterior, limpiaremos completamente el despliegue.

1. **Los servicios desplegados con stack son servicios de swarm, por lo que su gestión se realiza mediante service start|stop.**
2. **Para borrar una aplicación desplegada mediante stack, usaremos docker stack rm.**
3. **Docker stack elimina completamente el despliegue; podremos verificarlo revisando de nuevo las redes y volúmenes.**

```
$ docker volume ls && docker network ls
```

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- **Licenciamiento**
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito

Licenciamiento



Licenciamiento en Docker (por nodo. máx 2 cores físicos)

- Edición Comunitaria.
 - Engine y Registro Público (DockerHub).
- Edición Enterprise.
 - **Básica**
 - Soporte de Engine con Imágenes y Plugins Certificados y Registro Público (DockerHub).
 - **Estándar**
 - Básico + Docker Datacenter (Universal Control Plane + Docker Trusted Registry)
 - **Avanzada**
 - Estándar + Verificación de CVE de Imágenes (Scanning)

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- **Entorno Empresarial**
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- Casos de Éxito



Entorno Empresarial

Componentes y Modelos

- **Docker Hub**
- **Docker Cloud**
- **Docker Datacenter**
 - Engine
 - Universal Control Plane
 - Docker Trusted Registry

Entorno Empresarial



Docker Hub

- Registro Público de imágenes
 - Imágenes Públicas y Privadas (1 gratuita)
 - Búsqueda e Histórico de Capas
 - Agrupaciones de Usuarios y Organizaciones
 - Integración Out-Of-Box con Engine
 - Integración con Gestores de Código
 - Automatizaciones con Webhooks
 - Scanning de Imágenes Oficiales y Privadas
 - Firma de Imágenes

Entorno Empresarial



Docker Cloud

- Entorno completamente desplegado en la nube.
- Gestión de Cluster swarm.
- Integración de hosts mediante proveedores en la nube (AWS, Azure, etc...).
- Integración con repositorios de código online (github y bitbucket).
- Notificaciones mediante correo y slack de diferentes acciones.
- Utiliza Docker Hub como registro.

Entorno Empresarial



Universal Control Plane

- Gestión de Cluster Swarm
 - Integración Out-Of-Box
 - Gestión de todos los objetos del cluster
 - Swarm
 - Nodos
 - Servicios
 - Contenedores
 - Volúmenes
 - Redes
 - Secretos
 - Stacks y Aplicaciones

Entorno Empresarial



Universal Control Plane

- Gestión de Cluster Swarm (Continuación)
 - Autenticación y Autorización
 - LDAP / Directorio Activo
 - Acceso a recursos gestionado por etiquetas y grupos
 - Roles y Equipos (Grupos de usuarios)
 - Sin Acceso
 - Sólo visualización
 - Control Restringido
 - Control Total
 - Administración

Entorno Empresarial



Universal Control Plane

- Gestión de Cluster Swarm (Continuación)
 - Integración con Sistemas de Gestión de Logs
 - HTTP/S Routing Mesh
 - Integración directa con DTR
 - Acceso a estadísticas
 - Backup integrado

Entorno Empresarial



Docker Trusted Registry

- Registro Local de Imágenes
 - Se ejecuta sobre UCP
 - Alta disponibilidad mediante RAFT
 - Diferentes opciones de almacenamiento
 - HTTPS out-of-box e integración en clientes
 - Firma de imágenes con integración de Notary
 - Planificación de purgado de imágenes sin uso

Entorno Empresarial



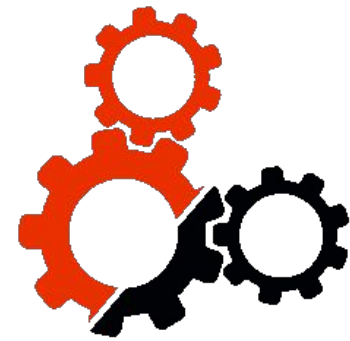
Docker Trusted Registry

- Registro Local de Imágenes (Continuación)
 - Autenticación y Autorización
 - LDAP / Directorio Activo
 - Roles de Acceso a Imágenes
 - Organizaciones
 - Backup integrado

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- **Contenedores en la vida del DevOps**
- Ejemplos cotidianos
- Casos de Éxito

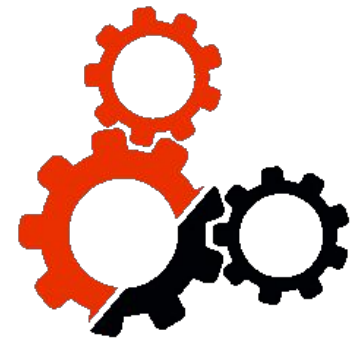
Contenedores en la vida del DevOps



DevOps - Integración Continua

- Automatización de la integración de imágenes base con código de aplicativo.
- Mediante triggers podremos seguir los cambios en los repositorios de código y generar nuevas imágenes.
- Despliegue de entornos dinámicos de compilación basados en esclavos para compilaciones distribuidas.

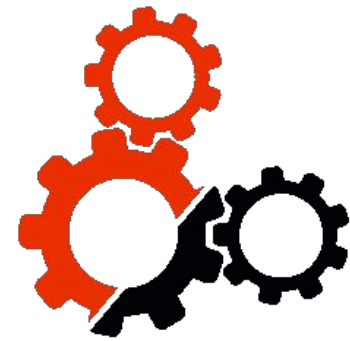
Contenedores en la vida del DevOps



DevOps - Integración Continua

- Firma de imágenes para garantizar entrega mediante flujo acordado en la organización.
- Integración con pruebas funcionales y de calidad de código basadas en ejecución en entornos aislados.

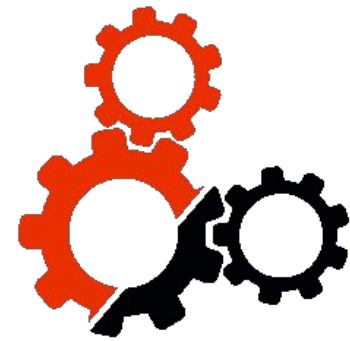
Contenedores en la vida del DevOps



DevOps - Despliegue Continuo

- El uso de Contenedores Software asegura que la aplicación desplegada es la misma que la desarrollada, salvo configuraciones.
- El firmado de imágenes permite asegurar los entornos productivos de ejecuciones de aplicaciones fuera del flujo acordado.
- El despliegue mediante contenedores permite actualización de componentes (microservicios) de forma muy sencilla.

Contenedores en la vida del DevOps



DevOps - Despliegue Continuo

- Recuperar el entorno de producción de una situación de errores tras un nuevo despliegue supone aplicar la imagen anterior.
- Las aplicaciones desplegadas sobre Swarm cuentan con “rolling update” para minimizar el impacto sobre el servicio.
- Reproducibilidad y gestión de versiones.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- **Ejemplos cotidianos**
- Casos de Éxito

Ejemplos Cotidianos



Empaquetado de aplicaciones para clientes

- Es el uso más sencillo de los entornos de contenedores software.
- Los clientes reciben imágenes de aplicativo, con versiones concretas de librerías, binarios, configuraciones, etc..., y estos simplemente ejecutan contenedores perfectamente operativos.

Ejemplos Cotidianos



Alta disponibilidad de aplicaciones

- Es el uso más sencillo de los entornos de contenedores software.
- Los clientes reciben imágenes de aplicativo, con versiones concretas de librerías, binarios, configuraciones, etc..., y estos simplemente ejecutan contenedores perfectamente operativos.

Ejemplos Cotidianos



Desarrollo local de código

- Los desarrolladores pueden usar sus propios equipos.
- Usando stacks o compose pueden desplegar entornos similares con pocos recursos.
- Se podrá verificar los aplicativos de forma sencilla antes de entrar en el flujo de desarrollo corporativo.

Ejemplos Cotidianos



Despliegue ágil de entornos de desarrollo

- En entornos de desarrollo ágiles resulta fundamental poder disponer de un entorno completo en el menor tiempo posible.
- La reproducibilidad de entornos con contenedores permite despliegues con caducidad para desarrolladores.
- Los recursos de estos entornos son fácilmente decomisables después de su uso.

Ejemplos Cotidianos



Aplicaciones desplegadas sobre servidores de aplicaciones

- Las aplicaciones desplegadas sobre servidores de aplicaciones están divididas → microservicios.
- Se sustituye cada despliegue sobre el servidor de aplicaciones por un contendor con la aplicación como servicio único.
- La integración entre aplicaciones se realiza mediante integración en herramientas de gestión de APIs o publicando cada servicio por separado.

Ejemplos Cotidianos



Despliegue de entornos multi-site

- Los entornos web multi-site presentan una publicación distribuida → microservicios.
- Cada site se presenta sobre su propio servidor web.
- Todos los sites pueden publicarse o enrutarse de forma homogénea usando un proxy también desplegado mediante contenedores.
- Es posible publicar los servicios usando HTTP Routing Mesh mediante host header.

Introducción a los Contenedores Software

- Qué son los Contenedores y Qué problemas resuelven?
- Hardware vs Virtualización vs Contenedores
- Conceptos Clave
- Herramientas
- Seguridad
- Almacenamiento de Datos
- Comunicaciones
- Orquestación
- Despliegue de Servicios y Aplicaciones
- Licenciamiento
- Entorno Empresarial
- Contenedores en la vida del DevOps
- Ejemplos cotidianos
- **Casos de Éxito**

Casos de Éxito



Casos de éxito

- Adopción de Docker para mejorar:
 - Gestión de Código
 - Time To Market
 - Calidad de Entregas Software
 - Seguridad y Aislamiento
 - Despliegue de Aplicaciones en diferentes infraestructuras

Casos de Éxito



Gestión de Código

- Arquitectura basada en Micro-Servicios es fácilmente convertible en contenedores.
- Focalización de grupos de desarrollo.
- Mejoras en Prototipado y Estandarización de componentes.
- Se tiene Versionado independiente para cada componente.
- Infraestructura reproducible.

Casos de Éxito



Time to Market

- Cambios eficaces sin perturbar otros componentes.
- Estabilidad de componentes de forma independiente.
- Actualización continua sin impacto en servicio.
- Solución de errores más sencilla por componente.
- Escalado de componentes de forma independiente bajo demanda de servicio
 - Actualizaciones independientes de arquitectura subyacente

Casos de Éxito



Calidad de Entregables

- Las firmas garantizan el workflow correcto.
- Eliminamos dependencias de proveedores.
- Actualización continua sin impacto en servicio.
- Responsabilidades sobre aplicación recaen en desarrollador e infraestructura sobre devops/producción.

Casos de Éxito



Seguridad y Aislamiento

- Entorno productivo sólo admite despliegues firmados.
- Menor superficie de ataque en los servicios.
- Aislamiento entre aplicativos.
- Los sistemas subyacentes no se ven comprometidos en caso de acciones malintencionadas sobre aplicativos desplegados en contenedores.
- Resiliencia para los despliegues.

Casos de Éxito



Despliegue de Aplicaciones en diferentes infraestructuras

- Infraestructura reproducible en cualquier entorno.
- Escalado de Componentes de forma independiente.
- Actualizaciones independientes de arquitectura subyacente.
- Despliegue independiente de infraestructura (Cloud, On-Premise, equipo desarrollador).
- Empaquetado de aplicaciones (soporte a clientes, IoT - Resin.io, etc...).

Casos de Éxito



Referencias:

<https://www.docker.com/customers>

- VISA

<https://blog.docker.com/2017/04/visa-inc-gains-speed-operational-efficiency-docker-enterprise-edition/>

- PayPal

<https://www.docker.com/customers/paypal-uses-docker-containerize-existing-apps-save-money-and-boost-security>

- BBC News

<https://www.docker.com/customers/bbc-news-reduce-ci-job-time-over-60>

Casos de Éxito



Referencias:

<https://www.docker.com/customers>

- eBay

<https://www.docker.com/customers/ebay-simplifies-application-deployment>

- General Electric

<https://www.docker.com/customers/ge-uses-docker-enable-self-service-their-developers>

- New York Times

<https://www.docker.com/customers/new-york-times-delivers-continuous-integration-pipeline-docker>

Casos de Éxito



Referencias:

<https://www.docker.com/customers>

- UBER

<https://www.docker.com/customers/uber-accelerates-developer-onboarding-weeks-minutes-docker>

- Spotify

<https://www.docker.com/customers/continuous-delivery-spotify>

- Universidad de Indiana

<https://www.docker.com/customers/indiana-university-delivers-state-art-it-115000-students-docker-datacenter>