**Arquitecturas SOA**

Una arquitectura basada en SOA debe seguir una serie de principios para ser exitosa. Estos **principios** son:

* Cada servicio debe ofrecer un contrato para conectarse con él. Un caso muy común es un servicio que ofrece una API REST. Dicha API debe siempre mantener compatibilidad con versiones anteriores, o gestionar versiones de sus endpoints cuando se producen incompatibilidades, pero es fundamental no romper el contrato con otros servicios.
* Cada servicio debe minimizar las dependencias con el resto. Para esto es fundamental acertar con el scope de un servicio. Una indicación de que el scope no es el adecuado es cuando se producen dependencias circulares entre los servicios.
* Cada servicio debe abstraer su implementación. Para el resto de servicios debe ser transparente si un servicio usa un backend u otro para la base de datos o si ha hecho una nueva release.
* Los servicios deben diseñarse para maximizar su reutilización dado que la reutilización de componentes es una de las ventajas de una arquitectura SOA.
* Cada servicio tiene que tener un ciclo de vida independiente, desde su diseño hasta su implantación en los entornos de ejecución.
* La localización física de donde corre un servicio debe ser transparente para los servicios que lo utilizan.
* En lo posible, los servicios deben evitar mantener estado.
* Es importante mantener la calidad de los servicios. Un servicio con continuas regresiones puede afectar a la calidad final percibida por el resto de servicios que hacen uso de él.

**Microservicios**

Teniendo en cuenta estos principios, el concepto de **microservicios** es un poco la manera que se ha puesto de moda para referirse a las arquitecturas SOA, pero incidiendo más aún en que la funcionalidad de dichos servicios debe ser la mínima posible. Una medida bastante extendida es que un microservicio es un componente que debería ser desarrollable en unas dos semanas. Las ventajas de una arquitectura basada en microservicios son las siguientes:

* Son componentes pequeños que agilizan los procesos de desarrollo de software y son fáciles de abordar por un equipo de desarrolladores.
* Son servicios independientes, si un microservicio falla no debería afectar a los demás.
* El despliegue de un microservicio a producción es más sencillo que el de una aplicación monolítica.
* Los microservicios son altamente reutilizables.
* Los microservicios son más fáciles de externalizar.

**Construcción de imágenes. Docker build y Dockerfile**

Como vimos en la introducción a Docker, una imagen se corresponde con la información necesaria para arrancar un contenedor, y básicamente se compone de un sistema de archivos y de otros metadatos como son el comando a ejecutar, las variables de entorno, los volúmenes del contenedor, los puertos que utiliza nuestro contenedor…

La manera recomendada de construir una imagen es utilizar un fichero Dockerfile, un fichero con un conjunto de instrucciones que indican cómo construir una imagen de Docker. Las instrucciones principales que pueden utilizarse en un Dockerfile son:

* FROM image: para definir la imagen base de nuestro contenedor.
* RUN comando: para ejecutar un comando en el contexto de la imagen.
* ENTRYPOINT comando: para definir el entrypoint que ejecuta el container al arrancar.
* CMD comando: para definir el comando que ejecuta el container al arrancar.
* WORKDIR path: para definir el directorio de trabajo en el contenedor.
* ENV var=value: para definir variables de entorno.
* EXPOSE puerto: para definir puertos donde el contenedor acepta conexiones.
* VOLUME path: para definir volúmenes en el contenedor.
* COPY origen destino: para copiar ficheros dentro de la imagen. También se usa para multi-stage builds.

Para una lista completa de las instrucciones disponibles ir a la [**documentación oficial**](https://docs.docker.com/engine/reference/builder/).

Un ejemplo de un dockerfile para una aplicación Flask en python podría ser:

FROM ubuntu:latest

RUN apt-get update -y

RUN apt-get install -y python-pip python-dev

WORKDIR /app

ENV DEBUG=True

EXPOSE 80

VOLUME /data

COPY . /app

RUN pip install -r requirements.txt

ENTRYPOINT ["python"]

CMD ["app.py"]

**La caché de Docker**

La construcción de una imagen de Docker dado un Dockerfile puede ser un proceso costoso ya que puede implicar la instalación de un número elevado de librarías, y al mismo tiempo es un proceso bastante repetitivo porque sucesivos *builds* del mismo Dockerfile suele ser similares entre sí. Es por eso que Docker introduce el concepto de la **cache** para optimizar el proceso de construcción de imágenes.

La primera optimización que hace la *cache* de Docker es la descarga de la imagen base de nuestro Dockerfile. Docker descargará la imagen base siempre que la misma no se encuentre ya descargada en la máquina que hace el *build*. Esta optimización parece obvia ya que estas imágenes pueden tener un tamaño de cientos de MB, pero hay que tener cuidado ya que si la versión remota de la imagen cambia, **Docker seguirá utilizando la versión local**. Por tanto, si queremos ejecutar nuestro Dockerfile con la nueva versión de la imagen base deberemos de hacer un docker pull manual de la imagen base, o ejecutar docker build --pull.

Como hemos comentado anteriormente, una imagen de Docker tiene una estructura interna bastante parecida a un repositorio de *git*. Lo que conocemos como *commits* en *git* lo denominamos *capas* de una imagen en Docker. Por lo tanto, una imagen (o repositorio) es una sucesión de capas en un Registro de Docker, donde cada capa almacena un *diff* respecto de la capa anterior. Esto es importante de cara a optimizar nuestros Dockerfiles, como veremos en la siguiente sección.

Por ahora bastará saber que cada instrucción de nuestro Dockerfile creara una y sólo una capa de nuestra imagen. Por lo tanto, la *cache* de Docker funciona a nivel de instrucción. En otras palabras, si una línea del Dockerfile no cambia, en lugar de recomputarla, Docker asume que la capa que genera esa instrucción es la misma que la ejecución anterior del Dockerfile. Por lo tanto, si tenemos una instrucción tal como:

RUN apt-get update && apt-get install -y git

que no ha cambiado entre dos *build* sucesivos, los comandos *apt-get* no se ejecutarán, sino que se reusará la capa que generó el primer *build*. Por tanto, aunque antes de ejecutar el segundo *build* haya una nueva versión del paquete *git*, la imagen construida a partir de este Dockerfile tendrá la versión de *git* anterior, la que se instaló en el primer *build* de este Dockerfile. Podemos desactivar el uso de la cache ejecutando docker build --no-cache.

Es importante destacar los siguientes aspectos sobre la *cache* de Docker:

* La *cache* de Docker es local, es decir, si es la primera vez que haces el build de un Dockerfile en una máquina dada, todas las instrucciones del Dockerfile serán ejecutadas, aunque la imagen ya haya sido construida en un Registro de Docker.
* Si una instrucción ha cambiado y no puede utilizar la *cache*, la *cache* queda invalidada y las siguientes instrucciones del Dockerfile serán ejecutadas sin hacer uso de la *cache*.
* El comportamiento de las instrucciones ADD y COPY es distinto en cuanto al comportamiento de la *cache*. Aunque estas instrucciones no cambien, invalidan la caché si el contenido de los ficheros que se están copiando ha sido modificado.

**Comandos comunes para gestionar imágenes**

Estos son los comandos más comunes para el manejo de imágenes:

* *docker build*: nos permite crear una imagen a partir de un Dockerfile..
* *docker login*: autentica la cli de docker contra un registro, por defecto Docker Hub.
* *docker pull*: descarga una imagen a la que tengamos acceso desde un registro.
* *docker image ls*: lista las imágenes que están disponibles en nuestra máquina.
* *docker inspect*: muestra información detallada de una imagen. Se puede acceder a un campo particular con el comando docker inspect -f '{{.Size}}' imagen.
* *docker image rm*: elimina una imagen.

**Buenas prácticas**

**1. Usa .dockerignore**

El build de una image se ejecuta a partir de un Dockerfile y de un directorio, que se conoce con el nombre de *contexto*. Este directorio suele ser el mismo que el directorio donde se encuentra el Dockerfile, por lo que si ejecutamos la instrucción:

ADD app.py /app/app.py

Estamos añadiendo a la imagen el fichero app.py del contexto, es decir, el fichero app.py que se encuentra en el directorio donde está el Dockerfile. Dicho directorio se comprime y se manda al Docker Engine para construir la imagen, pero puede que tenga ficheros que no son necesarios. Es por eso que este directorio puede tener un fichero .dockerignore, que de una manera similar a fichero .gitignore, indica los ficheros que no deben ser considerados como parte del contexto del *build*.

**2. Reduce el tamaño de tus imágenes al mínimo**

Tu imagen Docker sólo debe contener lo estrictamente necesario para ejecutar tu aplicación. Con el objetivo de reducir complejidad, dependencias, tamaño de las imágenes, tiempos de *build* de una imagen, debes evitar la instalación de paquetes sólo por el hecho de que puedan ser *útiles* para depurar un contenedor. Como ejemplo, no incluyas editores de texto en tus imágenes.

Otro opción muy práctica es el uso de imágenes base pequeñas, por ejemplo, haciendo uso de alpine.

Por ejemplo, compara docker-for-dev/flask-ubuntu y docker-for-dev/flask-alpine.

**3. Ejecuta sólo un proceso por contenedor**

Salvo raras excepciones, es recomendable correr sólo un proceso por contenedor. Esto permite reutilizar contenedores más fácilmente, que sean más fáciles de escalar, y da lugar a sistemas más desacoplados. Por ejemplo saca tu lógica de *logging* a un contenedor independiente.

**4. Minimiza el número de capas de tu imagen.**

Como hemos dicho anteriormente, cada capa de una imagen se corresponde con una instrucción del Dockerfile. Compare el Dockerfile:

RUN apt-get update

RUN apt-get install -y bzr

RUN apt-get install -y cvs

RUN apt-get install -y git

RUN apt-get install -y mercurial

con este otro:

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

apt-get clean

Ambos son igualmente legibles, pero el primero genera 5 capas, y el segunda sólo una, que además ejecuta un apt-get clean que reduce el tamaño de dicha capa. Recuerda que cada instrucción del Dockerfile genera una capa en la imagen final. Por tanto, si hacemos un *apt-get install* en una instrucción, y un *apt-get clean* en otra instrucción, habremos dejado una capa con todos ficheros que luego el *apt-get clean* borra.

##### 5. Optimiza el uso de la cache.

Optimiza el uso de la cache añadiendo al principio de tu Dockerfile las instrucciones que menos cambian (como la instalación de librerías), y dejando para el final las que más cambian (como el copiado del código fuente). Como ejemplo compare el Dockerfile:

Por ejemplo, compara docker-for-dev/flask-alpine y docker-for-dev/flask-build-cache.

El primero cachea la instalaciones de las dependencias pip siempre que no añadamos nuevas dependencias al fichero requirements.txt, antes de añadir el código fuente. Sin embargo, el segundo, aunque genere menos capas, no reusa la instalación de las dependencias porque ADD \* /app invalida la cache en cuanto hay un cambio en nuestro código fuente.

##### 6. Parametriza tus Dockerfiles usando argumentos

Aumenta la reusabilidad de tus Dockerfiles entre distintos entornos y aplicaciones parametrizando tus Dockerfiles con argumentos. Los argumentos son valores que se pasan como parámetros a cada build (aunque pueden tener valores por defecto), y que puedes utilizar en las instrucciones de tu Dockerfile. Por ejemplo, el Dockerfile:

FROM ubuntu

ARG user=root

ARG password

RUN echo $user $password

puede ser parametrizado de la siguiente manera:

docker build -t imagen --build-arg password=secret .

##### 7. Utiliza multi-stage builds

Los multi-stage es una funcionalidad introducida recientemente y que ayuda a crear imágenes muy pequeñas. Permiten resetear el sistema de ficheros de la imagen que se está construyendo, cambiar a otro sistema de fichero, pero importar ficheros de la imagen anterior.

Tenemos un ejemplo en docker-for-dev/go-multi-stage.

El primer FROM línea inicializa el sistema de ficheros con una imagen que lleva Go instalado. En esa imagen añadimos el directorio actual con todo su contexto y hacer el build de nuestro programa Go. Luego viene una nueva instrucción FROM que inicializa el sistema de ficheros con una imagen alpine sin nada instalado. La instrucción COPY copia el binario generado en el stage anterior y lo copia en la imagen actual. El resultado es una imagen muy pequeña, ya que no lleva el compilador de Go incluido, solo lleva el binario que necesitamos.

**Docker Compose y docker-compose.yml**

Docker compose es otro proyecto [**open source**](https://github.com/docker/compose) que permite definir aplicaciones muilti-contenedor de una manera sencilla y declarativa. Es una herramienta ideal para gestionar entornos de desarrollo, pero también para configurar procesos de integración continua en próximas lecciones.

docker-compose es una alternativa más cómoda al uso de los comandos docker run y docker build, que resultan un tanto tediosos cuando trabajamos con aplicaciones de varios componentes. Con Docker Compose se define un fichero docker-compose.yml que tiene esta forma (tomado de docker-for-devs/auto-build/docker-compose.yml):

web:

build: .

ports:

- "5000:5000"

depends\_on:

- redis

redis:

image: redis

donde estamos definiendo una aplicación que se compone de un contenedor definido desde un Dockerfile local, que escucha en el puerto 5000, y que hace uso de redis como un servicio externo. Dada esta definición, la manera de levantar la aplicación es simplemente:

docker-compose up -d

docker-compose acepta distintos comando, una lista completa puede encontrarse [**aquí**](https://docs.docker.com/compose/reference/).

**Comandos Docker Compose**

Destacar los siguientes puntos sobre docker-compose:

* docker-compose up -d levanta la aplicación en modo demonio, docker-compose up la levanta en primer plano, mostrando los logs de los distintos contenedores. La ejecución sucesiva del comando docker-compose up -d sólo recrea los contenedores que hayan cambiado su imagen o su definición. docker-compose up -d no hace el build cada vez que es invocado de las imágenes locales. Si deseas actualizar tu aplicación en base a los últimos cambios de tu código, tendrás que ejecutar docker-compose up --build -d. Un truco para mejorar este proceso es montar tu código como un volumen en el fichero docker-compose.yml, de tal manera que tu container siempre ve los últimos cambios en tu código fuente. Si quieres levantar solo uno o varios de los servicios en un compose, puedes añadir su nombre, por ejemplo docker-compose up -d redis.
* docker-compose pull actualiza las imágenes definidas en el compose con la versión actual que haya en el registro. En otras palabras, si alquien hace un push al registro, actualiza la versión de estas imágenes en nuestra máquina. Con la opción --parallel hace el pull en paralelo. Como todo comando de docker-compose se puede hacer pull de un subconjunto de servicios: docker-compose pull servicioA ServicioB.
* docker-compose build reconstruye las imágenes de los servicios que tengan una sección de build definida. Opciones interesantes son: --no-cache para invalidad la caché, --pull para hacer pull de las imágenes base y --build-arg key=val para pasar argumentos. Como todo comando de docker-compose se puede hacer build de un subconjunto de servicios: docker-compose build servicioA ServicioB.
* docker-compose push pushea al registro la versión local de las imágenes con una sección de build definida. Como todo comando de docker-compose se puede hacer pull de un subconjunto de servicios: docker-compose push servicioA ServicioB.
* docker-compose run ejecuta un contenedor de uno de los servicios definido en el compose. La diferencia principal con docker-compose up es que permite definir el comando a ejecutar, así como otra información de contexto como variables de entorno, el entrypoint, volúmenes, el directorio de trabajo… Es uno de los comandos más útil para el entorno del desarrollador. Por ejemplo, podemos definir un servicio en nuestro docker-compose.yml con todas las dependencias necesarios para ejecutar nuestros comandos de desarrollo. Haciendo uso de docker-compose run podemos ejecutar comandos aleatorios en ese entorno, evitando la necesidad de instalar todas las dependencias del entorno en la máquina actual. El caso más común es tener un servicio para ejecutar tests, como veremos más adelante, pero podríamos tener para cualquier tipo de tarea.
* docker-compose rm elimina los contenedores y otros recursos como redes, creados a partir de un compose.

docker-compose permite definir prácticamente todos los flags que soportan tanto el comando docker run como el docker build, pero docker-compose es mucho más fácil de utilizar. Las opciones más comunes son:

* build: para indicar que el container se construye desde un Dockerfile local. Puede tener subcampos como context, dockerfile, cache\_from o args.
* image: para indicar que el container corre un imagen remota. También indica el nombre de la imagen que se crea si hay un campo build.
* command: para redefinir el comando que ejecuta el container en lugar del comando definido en la imagen.
* environment: para definir variables de entorno en el contenedor. Se pueden pasar haciendo referencia a un fichero usando la propiedad env\_file. Si la variable no tiene un valor dado, su valor se cogerá del entorno de shell que ejecuta el docker-compose up, lo que puede ser útil para pasar claves, por ejemplo.
* depends\_on: para definir relaciones entre contenedores.
* ports: para mapear los puertos donde el contenedor acepta conexiones.

[**Aquí**](https://docs.docker.com/compose/compose-file/) tenéis ana lista completa y actualizada de las opciones que permite docker-compose.

**Volúmenes**

Cuando un contenedor es eliminado, la información contenida en él desaparece. Para evitar este problema y que los datos generados en el interior de un contenedor no se eliminen cuando el contenedor termina podemos hacer uso de volúmenes de datos (data volume). Un volumen es un directorio dentro del contenedor que se asociado con un directorio del host, por lo que persiste a la finalización del contenedor. Un contenedor puede tener varios volúmenes, y un mismo volumen puede montarse en varios contenedores para compartir información.

#### Volúmenes de Datos

Los volúmenes de datos tienen las siguientes características:

* Cuando borramos el contenedor, no se elimina el volumen asociado.
* Nos permiten guardar e intercambiar información entre contenedores.
* No son gestionados por los storage drivers, por lo que las operaciones de entrada / salida son mucho más eficientes.

Los volúmenes de datos tienes su propia interfaz con la línea de comandos de docker:

* docker volume create: crea un nuevo volumen de datos.
* docker volume ls: muestra los volúmenes de datos de nuestra máquina.
* docker inspect: devuelve información relativa a un volumen.
* docker volume rm: elimina un volumen de datos. También se pueden eliminar automáticamente al eliminar un contenedor si ejecutamos docker rm -f.

Si hacemos un docker inspect de un contenedor con volúmenes asociados esta información aparece en el campo Mounts.  
Por último podemos comprobar que aunque borremos el contenedor, el volumen no se borra.

#### Volúmenes del Host

Una aplicación particular de los volúmenes es la posibilidad de montar en el contenedor un directorio ya existente en el host. En este caso hay que tener en cuenta que si el directorio de montaje del contenedor ya existe, no se borra su contenido, simplemente se monta encima. Veamos un ejemplo:

$ docker run -it -v `pwd`:/data alpine sh

# cd data

# ls

#### Gestión de Volúmenes con docker-compose

El siguiente ejemplo ilustra la gestión de volúmenes con docker-compose:

version: '3.4'

services:

mysql:

image: mysql

volumes:

- mysql:/var/lib/mysql

- logs:/var/log/mysql

- /etc:/etc

mysql:

image: log-analizer

volumes:

- logs:/var/log:ro

volumes:

data:

logs:

**Redes**

Docker nos permite crear diferentes redes virtuales para nuestras necesidades, ya bien para unir o segmentar diferentes contenedores. De esta manera, podemos separar contenedores por seguridad en redes diferentes, o unirlos en la misma por conveniencia o por conectar sus servicios entre sí.

Por defecto, Docker nos ofrece tres tipos de redes diferentes. La primera, bridge, es donde arrancarían todos nuestros contenedores por defecto. Es una red que crea un puente entre la interfaz de red del contenedor que arrancamos y una interfaz de red virtual que se crea en nuestro equipo cuando instalamos Docker. La siguiente sería host. Host lo que hace es copiar la configuración de red del host, es decir, del servidor o máquina donde está Docker en el contenedor que estamos arrancando. Si arrancamos un contenedor aquí y ejecutamos la revisión de la configuración de red, veremos que es la misma que la de la máquina en la que lo estamos corriendo. Y después tenemos la red none, que utiliza el driver null, que lo que hace es eliminar toda la configuración de red de nuestro contenedor. Si creamos un contenedor aquí, solo tendremos loopback, solo tendremos la dirección 127.0.0.1, y no podremos conectar ningún sitio más.

Cuando instalamos Docker, veremos que nos crea una interfaz llamada docker0. Tiene una dirección IP privada, probablemente esta, si no da colisión con ninguna otra dirección IP que tengáis configurada. Y cuando creáis algún tipo de contenedor que se conecta a la red bridge, lo que hace es recibir por DHCP una dirección IP de este rango. Podéis conectar a través de él. Todos vuestros contenedores harán NAT a través de esta IP, y a través de la IP de salida de la máquina host o servidor en la que tenéis Docker instalado. Esta es vuestra red por defecto. De la misma manera, podéis conectar desde aquí a través de esta interfaz y por esta dirección IP a las direcciones IP de los contenedores que tenéis corriendo en esta red.

Esta red bridge no es la única que podéis tener, podéis crear más para separar vuestros contenedores en diferentes redes. Para ello, tendréis que ejecutar el siguiente comando: “docker network create”. Tenéis que elegir el driver del tipo de red que queréis crear. Lo más probable es que sea una red bridge. Y el nombre de la red. Veréis dos cosas, una que tenemos una red nueva con el nombre red1, driver bridge y un identificador. Otra, que si hacemos un ifconfig, vemos también que tenemos una interfaz virtual de red nueva en la que tenemos las siglas “br” de bridge y la ID de nuestra red. Si vemos esa interfaz en concreto, vemos que tenemos una nueva dirección IP. Dentro de este rango de red, aparecerán todos los contenedores que nosotros ejecutemos en red1. Y tanto red1 como bridge serán redes separadas. Los contenedores que tengamos en bridge, la red bridge por defecto, y los contenedores que tengamos en red1, la otra red bridge que hemos creado no se podrán comunicar entre sí.

Aparte, todo lo que arranquéis con Docker Compose, tendrá una red privada para él. Docker Compose creará una red cada vez que levantéis una infraestructura completa. Como veis, es bastante sencillo manejar las redes de Docker. Podéis segmentar todas las aplicaciones que corráis, por seguridad o por el sistema que utilicéis para conectarlas. Por último, podéis usar *docker-compose* para crear nuevas redes o lanzar los contenedores en redes específicas.

version: '3.4'

services:

proxy:

image: busybox

networks:

- outside

app:

image: busybox

networks:

- default

- inside

networks:

outside:

external: true

default:

inside:

driver: bridge

enable\_ipv6: true

Las redes tienen su propia interfaz con la línea de comandos de *docker*:

* *docker network create*: crea una nueva red.
* *docker network ls*: muestra las redes de nuestra máquina.
* *docker inspect*: devuelve información relativa a una red.
* *docker network rm*: elimina una red.

**Anexo**

**Repositorio:** <https://github.com/pchico83/docker-for-devs.git>