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… El *build* de una imagen termina una vez que la imagen de docker se sube a un Registro de Docker, momento en el cual comienza el periodo de despliegue de la aplicación.

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.
- `CMD comando`: para definir el comando que ejecuta el container al arrancar.
- `EXPOSE puerto`: para definir puertos donde el contenedor acepta conexiones.
- `ENV var=value`: para definir variables de entorno.
- `COPY origen destino`: para copiar ficheros dentro de la imagen.
- `VOLUME path`: para definir volúmenes en el contenedor.

Para una lista completa de las instrucciones disponibles ir a la documentación oficial.

La cache 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.

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 2 *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.

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.

Por último, si por algún motivo deseas hacer un *build* sin usar la *cache*, puedes hacer uso del flag `--no-cache=true` para dicho fin.

Consejos para escribir un Dockerfile

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.

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 \
```

```
    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.

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:

```
FROM python:2.7

WORKDIR /app

ADD requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

ADD * /app

CMD ["python", "app.py"]
```

con este otro:

```
FROM python:2.7

WORKDIR /app

ADD * /app
RUN pip install -r requirements.txt

CMD ["python", "app.py"]
```

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 .
```