[Docker SecDevOps] Capítulo 2 : Dockerfile

- # comentarios o directivas
- `INSTRUCCIÓN argumentos` : por convención, instrucción en mayúsculas
- primera instrucción: FROM (o ARG)

build

- `docker build` o `docker image build`
  - `-t <nombre_imagen>[:tag]`
  - `-f <nombre_fichero_Dockerfile>`

directivas

- antes de la instrucción **FROM**
- no repeticiones
- formato concreto: `# directiva=valor` (respetando espacios) → si no, es tratado como un comentario
- directivas soportadas actualmente:
  - **escape** : caracter de escape en ficheros Dockerfile. Soporta \ y `

ENV

- variables de entorno
- `ENV var=valor`
- `ENV var=valor var2=valor2 var3=$var2` ← produce una única capa de caché
- `ENV var valor`
- se referencian con el signo **$** o **${}**
- funcionalidades tipo bash:
  - ${var:-texto} : si var tiene valor propio (está inicializada) lo devuelve, si no, devuelve *texto*
  - ${var:+texto} : si var tiene valor propio, devuelve la cadena *texto*, si no, devuelve vacío
- se pueden usar en:
  - ADD
  - COPY
  - ENV
  - EXPOSE
  - FROM
  - ONBUILD
  - LABEL
  - STOPSIGNAL
  - USER
  - VOLUME
  - WORKDIR
- son de tipo global (afecta a todas las imágenes que desciendan donde fueron definidas)
- también llegan al contenedor
- se pueden sobrescribir con el parámetro **–env** en **docker run**

.dockerignore

- se procesa al mismo tiempo que se procesa el contexto en el **build** de una imagen
- ignora todos los archivos / directorios que estén especificados
- uso de comodines: *****, **?**, **!**
- comentarios: **#**
- importancia del orden de criterio de exclusión:

```
*.md
!README.md
```

```
*.md
!README*.md
README-secret.md
```

```
*.md
README-secret.md
!README*.md
```

    - el primer ejemplo excluye todos los ficheros *.MD* excepto el *README.md*
    - el segundo excluye todos los ficheros *.MD* excepto los *README*.md*, aunque el *README-secret.md* también quedaría excluido
    - el tercer ejemplo es una mala construcción por el orden de las instrucciones, ya que el fichero *README-secret.md* quedaría incluido, cuando lo que pretendemos es excluirlo
- se puede excluir *Dockerfile* también, pero solo se ignorará en las instrucciones **COPY** y **ADD**

FROM

- `FROM <imagen>[:tag|@digest] [AS <nombre>]`
    - si no se especifica *tag* se usará **latest**
    - el digest es el SHA256 de la imagen: `FROM busybox@sha256:3e8…0e7`
- primera instrucción del *Dockerfile* (con excepción de **ARG**)

multistage

uso de más de una imagen Docker para realizar la tarea

- uso de 2 o más FROM en el Dockerfile
- la imagen del último FROM es la que prevalece, todas las anteriores son descartadas
- es posible «traspasar» ficheros de un fase a otra con un parámetro en el comando **COPY**
    - `COPY –from=0 ….`
    - 0 haría referencia a la primera imagen usada, también se puede hacer referencia a través del nombre asignado en **AS**

RUN

ejecución de comandos en la imagen que estamos construyendo

- `RUN <comando>` → comando es pasado como parámetro a la shell del sistema:
    - linux: /bin/sh -c
    - windows: cmd /s /c

- `RUN [«ejecutable»,«parámetro1»,«parámetro2»]`
  - no se ejecuta shell (o para cambiar la shell o entornos sin shell)
  - vector JSON (comillas obligatorias)
- cada RUN genera una layer(capa)
- uso de | (pipe) para redirigir la salida de un ejecutable a otro
  - tener en cuenta que si falla la ejecución del primero, pero no del segundo, la ejecución se dará por buena
  - se puede usar `set -o pipefail` para evitar este comportamiento (aunque no todos los shell lo soportan)
  - `RUN [«/bin/bash», «-c»,«set -o pipefail && wget …»]`

CMD

ejecución en tiempo de creación del container

- proveer de valores por defecto
- los parámetros se pasarían a ENTRYPOINT
- entre esos valores se puede incluir un ejecutable
- 3 formas:
  - `CMD [«ejecutable»,«param1»,«param2»]`
  - `CMD [«param1»,«param2»,«param3»]` ← parámetros añadidos a ENTRYPOINT (también debe estar expresado como vector JSON)
  - `CMD comando param1 param2` ← el comando se ejecuta a través de la shell
- para asegurarse la ejecución de un programa hay que combinar ENTRYPOINT con CMD
- si se pasan parámetros en el **docker run**, estos sobreescriben los especifiados en el CMD

ENTRYPOINT

es el comando recomendable para definir el comando principal de una imagen

- 2 formas:
  - ENTRYPOINT [«ejecutable»,«param1»,«param2»]
  - ENTRYPOINT comando param1 param2 ← comando ejecutado a través de la shell
- es posible sobrescribir el ENTRYPOINT de una imagen a través del parámetro **--entrypoint** en **docker run**
  - `docker run --entrypoint «/bin/ls» debian -al /root`
    - -al y /root son parámetros pasados al nuevo **entrypoint**
- importante que la última instrucción ejecutada por el ENTRYPOINT se convierta en en el proceso con PID 1 del contenedor (a través de la instrucción **exec**) para que reciba las señales Unix enviadas al contenedor.

```
#!/bin/bash
set -e
if [ "$1" = 'postgres' ]; then
  chown -R postgres "$PGDATA"
  if [ -z "$(ls -A "$PGDATA")" ]; then
    gosu postgres initdb
  fi
  exec gosu postgres "$@"
fi
exec "$@"
```

- como regla general, debemos tener en cuenta:

    - cada Dockerfile debe tener definido un CMD o ENTRYPOINT
    - cuando queremos usar un contenedor como un fichero ejecutable, debemos usar ENTRYPOINT
    - CMD se debería usar para definir los parámetros por defecto para ENTRYPOINT o para ejecutar un comando de apoyo para la creación del contenedor, pero no el comando que ejecuta el proceso final del mismo

LABEL

añade metadatos a una imagen

- `LABEL key=value [key2=value2]`
- se pueden crear varias etiquetas o una única separando valores
- etiquetas con el mismo nombre, prevalece la última

EXPOSE

indica puertos y protocolos donde escuchará el contenedor

- con **docker run** podemos mapear los puertos con:
    - `-p host:contenedor`
    - `-P` : mapea los puertos indicados en EXPOSE a puertos no privilegiados aleatorios
    - los contenedores que comparten red, no necesitan mapear puertos, tienen acceso a todos ellos.

ADD

copia ficheros, directorios o ficheros remotos al directorio de destino en la imagen docker

- `ADD <src>… <dest>`
- `ADD [«<src>»,… «<dest>»]` : obligatorio en el caso de que algún elemento contenga espacios
- origen puede ser absoluto o relativo al contexto
    - si es un directorio, copiará el contenido
- destino puede ser absoluto o relativo al WORKDIR
    - si acaba en **/** copia el fichero origen con el mismo nombre
- origen permite caracteres comodín usando las reglas de filepath.Match del lenguaje Go: *****, **?**
- si origen está en un formato de compresión reconocido (gzip,bzip,xz) se descomprime automáticamente en destino
    - si origen es una URL a un archivo comprimido, no se descomprime, solo se copia.
- si se especifican múltiples orígenes o comodines, destino ha de ser directorio (y acabar en /)
- si destino no existe, se creará, con los directorios intermedios necesarios
- todos los ficheros y directorios serán creados con UID/GID 0
- en caso de URLs a ficheros remotos, los permisos se establecen a 600
    - ADD no tiene sistema de autentificación implementado, se debería usar **RUN**

COPY

copiar ficheros y directorios

- `COPY <src>… <dest>`
- `COPY [«<src>»… «<dest>»]`
- NO descomprime archivos

- NO copia ficheros remotos
- Permite copiar ficheros entre imágenes (multistage)
    - `COPY –from=…`

VOLUME

crea un punto de montaje con el nombre dado

- `VOLUME <path> [<path>…]`
- `VOLUME [«<path>» [, «<path>»]]`
- Los volúmenes se montan en tiempo de ejecución del contenedor
- en ese momento se le puede indicar a Docker donde montarlo:
    - con el parámetro `-v` o `–volume` o `–mount` especificando un directorio local
    - sin parámetro, ubica el volumen en **/var/lib/docker/volumes/<nombre_volumen>/_data**
        - para averiguar el nombre del volumen asociado:

```
docker container inspect 76ce590930b0 --format "{{ .Mounts }}"
```

        - en mac, el punto de montaje no está en la máquina Apple, si no en el sistema de ficheros de la máquina virtual que Docker usa por debajo

USER

establece el usuario (UID) y grupo (GID) del usuario que ejecuta los comandos de las instrucciones **RUN**, **CMD**, **ENTRYPOINT**

- `USER <user>`
- `USER <UID>`
- `USER <UID>[:<GID>]`
- por defecto se ejecutan como **root**
- se puede usar más de una vez en el mismo **Dockerfile**, aunque se recomienda minimizar su uso por su repercusión en las capas de la imagen
- también se puede usar **sudo**, aunque se desaconseja su uso (por temas relacionados con el envío de señales y la emulación de termianles) → recomendado **gosu**
- también se puede especificar el usuario en el momento de ejecución del contenedor con el parámetro `-u` o `–user`

WORKDIR

establece el directorio de trabajo para los comandos **RUN**, **CMD**, **ENTRYPOINT**, **COPY**, **ADD**

- `WORKDIR <path>` ← sin / final
- el directorio es creado inmediatemente
- pueden ser absolutos (recomendado) o relativos (al último **WORKDIR**)

ARG

variables que el usuario puede usar durante el **build** de la imagen

- `BUILD var=valor`

- también se pueden pasar como parámetro en **docker build** con `–build-arg var=valor`
- solo sobreviven *en la fase* en las que son definidas
- ENV tiene preferencia sobre ARG
- existe un conjunto de variables definidas:
    - HTTP_PROXY, http_proxy
    - HTTPS_PROXY, https_proxy
    - FTP_PROXY, ftp_proxy
    - NO_PROXY, no_proxy
    - que tienen las siguientes características:
        - se pueden pasar por parámetro al construir la imagen **–build-arg**
        - no se incluyen como parte del histórico de capas de la imagen, a menos que se defina explícitamente en **Dockerfile**

ONBUILD

trigger que se ejecuta cuando la imagen es usada como base para otra imagen

- `ONBUILD <instrucción>`
- no se pueden anidar 2 ONBUILD
- se pueden poner varios ONBUILD y serán ejecutados en el mismo orden
- se aplica en el primer nivel «de herencia»
- reutilización de una imagen «base»

STOPSIGNAL

define la señal que se enviará al contenedor cuando este se pare

HEALTHCHECK

comunicar el estado de un contenedor en tiempo de ejecución

- `HEALTHCHECK NONE` : deshabilita **healthcheck** heredados
- `HEALTHCHECK [OPCIONES] CMD comando`
    - opciones:
        - `–interval=ns` : segundos con los que se ejecuta el comando de chequeo. 30s por defecto
        - `–timeout=ns` : tiempo de espera para la ejecución del chequeo. 30s por defecto.
        - `–start-period=ns` : tiempo que necesita el contenedor para arrancar. Los chequeos en este período no se consideran erróneos
        - `–retries=N` : número de reintentos antes de considerar el contenedor como fallido
    - comando:
        - comando de terminal o array JSON (como en CMD/ENTRYPOINT)
        - el código de salida indica el estado del contenedor:
            - 0 - success
            - 1 - unhealthy
            - 2 - reserved (no usar)
        - la salida del comando - en fase de depuración - se puede consultar a través de `docker container inspect` (4096 bytes)
- solo puede haber 1 HEALTCHECK en el fichero **Dockerfile**, así que el último prevalece (permitiendo sobreescribir alguno heradado)

SHELL

indica que shell será usada por defecto

- `SHELL [«ejecutable»,«parámetros»]`
- por defecto:
  - LINUX: `SHELL [«/bin/sh»,«-c»]`
  - WINDOWS: `SHELL [«cmd»,«/S»,«/C»]`
- se puede usar varias veces en el **Dockerfile** y las instrucciones que precisen de un shell usarán el establecido en ese momento
- afecta a **CMD**, **ENTRYPOINT**, **RUN**

From:
https://matebcn.eu/ - **miguel angel torres egea**

Permanent link:
https://matebcn.eu/info:libros:docker-sec-dev-ops:cap2

Last update: **31/01/2019 09:44**