Despliegue de aplicaciones web

Tema 5. Contenedores y volúmenes

1. Introducción

Los datos que pueden alojarse en un contenedor, pueden tener diferente origen y condición:

- La aplicación en sí y su entorno de ejecución, cuyo código será inmutable una vez generada la imagen.
- Datos, información generada mientras el contenedor está en ejecución. Por ejemplo, datos introducidos por el usuario. Este tipo de información puede almacenarse durante la ejecución del contendor pero desparecerá al terminar la misma. Se trata de información temporal almacenada en el contenedor.
- Datos permanentes, que se almacenarán en ficheros de texto, bases de datos, etc. Se trata de información que no debería perderse si el contenedor se para. Para almacenar ese tipo de información, en esta unidad se va a ver el uso de **volúmenes**.

Ya se ha visto que las imágenes se forman a partir de capas. Los contenedores crean una capa extra sobre la imagen que utilizan y es en ella en la que escriben y modifican cosas, pero una vez el contenedor desaparece, esa capa desaparece también.

Si por ejemplo una aplicación almacena datos en una base de datos, y se actualiza la imagen por una actualización de versión, el contenedor anterior ya no podría usarse y debería arrancarse uno nuevo. Todos los datos almacenados en la base de datos del contenedor primero desaparecerían.

2. Volúmenes

Los volúmenes son carpetas en el disco duro de la máquina anfitriona que se montan, se mapean, en los contenedores. Es decir, permiten conectar carpetas localizadas fuera del contenedor con carpetas incluidas en el contenedor.

La aplicación de un contendor en ejecución podrá almacenar datos en un volumen y, al eliminarse el contenedor, el volumen no desaparecerá con el sino que permanecerá y con el los datos que almacene.

Si se arranca un nuevo contenedor y se monta en él un volumen existente, cualquier dato que el volumen almacene estará disponible en ese nuevo contenedor.

Los volúmenes son de lectura-escritura y son objetos de Docker, igual que los contenedores, las imágenes o etiquetas, y se puede trabajar con ellos de forma aislada.

Es Docker quien almacena las carpetas creadas en una ubicación de la máquina local no conocida por el usuario, ya que este tipo de directorios no están pensado para ser accedidos.

2.1. Volúmenes anónimos

Una forma sencilla de declarar un volumen es en el fichero Dockerfile con la instrucción **VOLUME**, indicando la carpeta, o carpetas, interna del contenedor que se quiera mapear:

```
VOLUME [“rutaRelativa”]
```

Un volumen declarado de esta forma no tendrá nombre y, al desaparecer el contenedor, el volumen desaparecerá con él siempre y cuando el contenedor se haya arrancado con la opción --rm. Si no es así, el volumen quedará “huérfano” y habrá que eliminarlo de propio intento.

docker volume

Con las opciones que ofrece este comando se puede tener acceso a información de un volumen anónimo. Por ejemplo, con **docker volume ls** se pueden ver los volúmenes existentes.

Otra opción que puede ser de utilidad, es **docker volume prune**. Con este comando se eliminarán los volúmenes no utilizados.

2.2. Volúmenes con nombre

Un volumen de este tipo si permanece tras la eliminación del contenedor, por lo que son una buena solución para los datos que deben persistir. Aun así, no servirían para almacenar datos que necesiten ser editados o accedidos directamente, ya que siguen estando en una ubicación desconocida.

Un volumen con nombre no se indica en el fichero Dockerfile sino que se declarará al arrancar el contenedor incluyendo la opción -v:

```
-v nombreVol:ubicacionInterna
```

Donde el nombre del volumen lo elige el usuario, y la ubicación interna es el path interno del directorio que se quiera persistir.

Si se quiere utilizar el mismo volumen para otro contenedor, bastará con indicar el mismo nombre de volumen.

EJERCICIO 1

- En el Aula Virtual encontrarás el proyecto "Ejercicio1". Descárgalo y mira que hace el código.
  - Crea un fichero Dockerfile que(ten cuidado con las rutas):
    - tome como imagen base java.
    - Copie todo el contenido de la carpeta src a /var/www/java.
    - Compile la aplicación.
    - E incluya la directiva para la ejecución de la aplicación cuando se arranque un contenedor.
  - Genera una imagen y ejecuta un contenedor para poder probar el programa. Comprueba que datos almacena el contenedor parándolo y arrancándolo de nuevo.
  - Elimina el contenedor y crea otro nuevo. Comprueba ahora si la información almacenada por el primer contenedor se muestra con la nueva información. Tras la comprobación elimínalo.
- Ejecuta un nuevo contenedor indicando:
  - Que el contenedor se eliminará cuando pare la ejecución.
  - Un volumen de nombre volumen1 en el que se persistirá la información almacenada en la carpeta BBDD.
  - Cuando termine la ejecución comprueba que volumen1 sigue existiendo.
- Arranca, ahora, otro contenedor en las mismas condiciones que el anterior.
  - Comprueba que, justo antes de que se termine la ejecución, las letras que se muestren sean las almacenadas por dicho contenedor y por el ejecutado antes que el.

3. Binding Mounts

Otra opción que Docker provee para poder persistir datos son los binding mounts. Consisten en montar ubicaciones conocidas del host dentro del contenedor.

Su funcionamiento es similar al de los volúmenes, con algunas particularidades:

- Los volúmenes los maneja Docker mientras que los bind mounts los maneja el usuario, por lo que la localización de los mismos dentro de la máquina host será conocida.
- El montaje sobre un directorio del contenedor que no está vacío, reemplaza el contenido original del contenedor.
- Los cambios ejecutados desde el contenedor sobre los archivos del bind mount reemplazan, por defecto, los archivos originales del host. Tanto Docker como el host están sincronizados en el acceso a una determinada ruta.
- Es un sistema adecuado para almacenar datos persistentes que deban ser accesibles para el usuario.

Por ejemplo, pueden ser útiles durante la fase de desarrollo de un proyecto para poder hacer modificaciones dentro de un contenedor y así no tener que rehacer la imagen cada vez que se realice algún cambio de código.

3.1. Crear un bind mount

Un binding mount se crea, al igual que hemos visto con los volúmenes con nombre, al ejecutar el contenedor.

La opción es la misma vista para volúmenes, -v, pero en lugar de indicar un nombre de volumen se indicará la ruta(absoluta) del host donde se quiera almacenar:

```
-v rutaExterna:rutaInterna
```

Al indicar la ruta externa se ha de tener cuidado con espacios y demás, así que es una buena practica incluir todo el par rutaExterna:rutaInterna entre comillas.

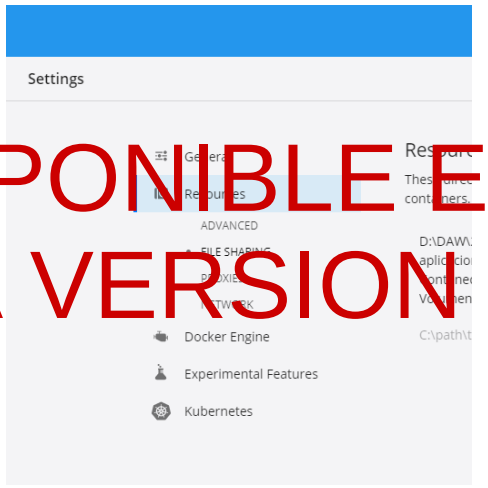
EJERCICIO 2

Partiendo del proyecto resultante del ejercicio 1:

- Ejecuta un contenedor de tal forma que, en lugar de crear el volumen volumen1 anterior, cree un bind mount para que los cambios que se produzcan en letras.txt se vean en la carpeta BBDD del proyecto local y que, si se hace alguna modificación en local se vea en el resultado final de la ejecución(por ejemplo si se borran las letras almacenadas hasta el momento).

3.2. Acceso a la carpeta local

Es posible que Docker no tenga acceso a la carpeta que se está compartiendo como un bind mount. Para asegurar el acceso, se va a la configuración de Docker en Settings->Resources->File Sharing:



Si el directorio(o alguna carpeta padre del mismo) que se quiere compartir no está en el listado mostrado, se puede añadir.

3.3. Combinar y unir volúmenes

Hay que tener cuidado cuando, haciendo uso de binding mounts, se mapea toda la carpeta local del proyecto con el directorio de trabajo del contenedor. Las dependencias, generaciones y demás que se establecen al construir la imagen, no están en la carpeta local del proyecto(por ejemplo los .class en un proyecto java, la carpeta node_modules en un proyecto nodeJS, etc.) y, al mapear todo el directorio de trabajo, esos elementos que no están en local se eliminan en el contenedor al sobre escribir , por el mapeo, todo el directorio de trabajo.

Para evitar eliminar contenido necesario, es posible crear, en el arranque del contenedor(o en Dockerfile), volúmenes anónimos para almacenar el contenido que no se quiera mapear.

Docker, cuando hay más de un volumen establecido para una misma ruta, aplica la mas restrictiva(la que tenga el path interno más largo).

Por ejemplo, se quiere mapear todo el directorio de trabajo de un proyecto nodeJS cuyo directorio raíz en el contenedor es /app. El bind mount se indicaría como sigue:

```
-v /user/directorio/proyecto:/app
```

En esa carpeta app, estará también la carpeta generada node_modules, por lo que se sobrescribirá en el bindeo. Para evitarlo, se incluiría en el arranque del contendor un volumen ANÓNIMO para dicho directorio:

```
-v /app/node_modules
```

3.4. Opciones de manejo de volúmenes

Las opciones que ofrece Docker para poder manejar los volúmenes son:

- docker volume create: Crear un volumen que después podrá referenciarse al ejecutar un contenedor.
- docker volume inspect: Muestra información detallada sobre uno o mas volúmenes.
- docker volume ls: El ya conocido comando para listar lo volúmenes existentes.
- docker volume prune: También visto anteriormente, permite eliminar todos los volúmenes que no estén en uso en el momento de la eliminación.
- docker volume rm: Comando para borrar uno o mas volúmenes

4. .dockerignore

Cuando se crea una imagen mediante el comando *docker build* a partir de un Dockerfile, todos los archivos y carpetas proporcionados en el path componen el contexto. La imagen de Docker se creará con este contexto, por lo que los archivos no deseados enviados al demonio se pueden empaquetar por error en la imagen final con las instrucciones ADD o COPY (Por ejemplo, al hacer COPY . .).

Además, el demonio de Docker se puede ejecutar en una máquina remota, por lo que es mejor evitar enviarle archivos grandes, innecesarios y sensibles.

Es posible definir un archivo .dockerignore, el cual la CLI de Docker busca cuando se invoca *docker build* en la carpeta raíz del contexto. Si este archivo existe, todos los archivos y carpetas que coincidan con los patrones de exclusión no se incluirán en el archivo enviado al demonio de Docker. La compilación de Docker será más rápida y se reducirá el riesgo de empaquetar archivos no deseados.

Por ejemplo, si desea excluir la carpeta .git, o el Dockerfile, se incluirá en el fichero .dockerignore:

```
Dockerfile
.git
```

5. Argumentos y variables de entorno

Docker soporta argumentos en tiempo de compilación y variables de entorno para tiempo de ejecución.

Variables de entorno

Al contrario que los argumentos los valores incluidos en las variables de entorno si serán accesibles desde el código de la aplicación.

Se pueden crear dentro del fichero Dockerfile con la instrucción ENV:

ENV nombreVariableEntorno valor

Para poder referenciarlas después se usará el nombre dado precedido del símbolo $.

También se puede crear mediante la opción --env(o -e) al ejecutar en contenedor, añadiendo tantas variables de entorno como se quiera:

-e variable=valor

Muchos lenguajes ofrecen la opción de recuperar mediante un objeto o similar las variables de entorno.

EJERCICIO 3

- Partiendo del proyecto "Ejemplo Docker NodeJS" de la primera unidad de Docker, realiza en código los cambios necesarios para que el puerto en el que escucha el servidor se establezca a través de una variable de entorno:
  - en nodeJS el puerto se recuperará con *process.env.PORT*, por lo que la variable que habrá que establecer es PORT.
- Modifica el fichero Dockerfile:
  - Incluye la variable con el valor 80.
  - Modifica la instrucción EXPOSE de acuerdo a la nueva variable.
  - Rehaz la imagen.
  - Comprueba que funciona la aplicación al ejecutar un contenedor.
- Sin modificar Dockerfile, ejecuta un segundo contenedor añadiendo la variable PORT al docker run. En este caso, asigna a la variable de entorno el puerto 8000. ¿Qué otro cambio deberás realizar en la sentencia de arranque del contenedor para que funcione la aplicación?

Para incluir mas de una variable de entorno y que la sentencia de arranque no crezca demasiado, es posible incluir todas las variables de entorno en un fichero .env en el directorio local(a la par que .dockeringnore), indicándolas con el formato var=valor. Este fichero se referenciará al arrancar el contenedor como sigue:

--env-file pathFichero

Por ejemplo: --env-file ./.env

Esta última opción puede ser interesante cuando se trata de valores que, por seguridad, no se quiere que sean vistos por cualquiera. Al no ir dentro de la imagen no podrán ser accesibles por otras personas.

EJERCICIO 4

- Prueba esta última opción en el proyecto del 3.

Argumentos

Variables que permiten asignar datos flexibles en el fichero Dockerfile, usados para poder incluir diferentes valores en algunas instrucciones del Dockerfile.

Para crear un argumento dentro del fichero Dockerfile se utiliza la instrucción ARG:

ARG nombreArg=valorDefecto

Y, para utilizar dicho argumento dentro de Dockerfile, se referenciará con el símbolo $ de nuevo.

Estos valores se pasarán al fichero Dockerfile cuando se vaya a construir la imagen con docker build, pasándolos con la opción --build-arg:

--build-arg nombreArg=valor es solo para el dockerfile, no funciona en cmd

Estos valores no serán accesibles desde CMD(porque es un comando para tiempo de ejecución) ni desde código de la aplicación.

EJERCICIO 5

Partiendo del resultado del ejercicio anterior haz las modificaciones necesarias para que:

- El valor del puerto que se va a exponer sea un valor que se pasará como argumento al crear la imagen.
- Sigue teniendo que haber una variable de entorno con dicho valor ya que se usa en código pero, en este caso, el valor de la variable de entorno se asignará a partir del argumento pasado al crear la imagen.
- Revisa que tus instrucciones estén en un orden óptimo.