

Laboratorio de Containers

Ferreyra, Martinez Castro, S. Rodríguez

Junio, 2019

1 Laboratorio de Docker

Esta es una primera aproximación a docker donde primero crearemos un container individual desde una imagen bajada desde el servidor de docker, oportunamente llamado DockerHub, para luego crear nuestra propia imagen con el código provisto y crear un container con este.

1.1 Levantar un container para ejecutar una tarea

Vamos a correr el comando `ls` desde un container con `alpine` que es una distribución de linux para ver los contenidos de la raíz en el container.

Para eso vamos a correr `'docker container run alpine ls'`.

Ese último comando contiene varias cosas. `'docker'` es el comando principal que se escribe para pasarle instrucciones a docker. Una de esas instrucciones es `'container'` que permite administrar los containers y dentro de esa instrucción podemos especificar una subinstrucción para administrarlos que es `'run'`.

`'run'` básicamente corre una instrucción en un nuevo container. De esta forma `'run'` simultáneamente crea un nuevo contenedor con la imagen que le pasamos como parámetro `'alpine'` y corre una instrucción que le pasamos como segundo parámetro, en este caso `'ls'`.

Luego de correr este comando vamos a ver que docker bajo la imagen oficial desde la página y que creo el container para por último correr el comando `ls` y mostrar la estructura del directorio en la raíz del container.

Podemos ver corriendo `'docker container ls -all'` que nuestro container sigue vivo y con un estado que nos indica que ya no se está utilizando. Además, podemos ver que nuestro container tiene un id y una imagen asociados; en este caso la imagen es `alpine`.

1.2 Correr un container interactivo de Ubuntu

Para este ejemplo vamos a correr una sesión de `bash` desde un container levantado con una imagen de `ubuntu`. Para eso vamos a utilizar nuevamente el comando `'run'` del ejemplo anterior con un par de parámetros extras.

Vamos a correr `'docker container run -interactive -tty -rm ubuntu bash'`.

Este comando utiliza los parámetros 'interactive', 'tty' y 'rm'. Estos parámetros en orden son: - Interactive relaciona el input producido desde el teclado del host con la entrada del container para la sesión de bashdocker classroom - Tty muestra, como estamos acostumbrados, el usuario actual tanto como el directorio haciendolo mas parecido aun a una terminal corriente - Rm elimina el container luego de terminar la sesion. De esta forma no se vera mas al correr 'docker container ls -all'

Luego de eso podemos correr un par de comandos sobre el nuevo container que instanciamos: - 'ls /' nos mostrara lo mismo que previamente hicimos para alpine - 'ps aux' nos muestra todos los procesos corriendo actualmente en el contenedor - 'cat /etc/issue' nos va a mostrar la distribución de linux sobre la que estamos

Este ultimo ejemplo es especialmente util ya que de esta manera podemos ingresar a un contenedor, configurar todo el entorno para nuestra aplicacion, administrar las dependencias requeridas y testear la aplicacion; y una vez que tenemos el contenedor como queremos podemos generar una imagen desde este y de esta forma distribuir libremente la imagen para que puedan utilizar tu aplicacion y tener el mismo container.

1.3 Correr un contenedor en background

Esta va a ser la manera en que vamos a correr la mayoría de los containers.

Vamos a correr un container con MySQL para luego explicar las opciones utilizadas:

```
docker container run \
--detach \
--name mydb \
-e MYSQL_ROOT_PASSWORD=1234 \
mysql:latest
```

En el comando anterior, ademas de añadir una etiqueta a la imagen que queriamos correr (mysql:latest, que nos trae la ultima version de MySQL), utilizamos opciones nuevas, que son:

- 'detach', pone a correr el servidor en background por lo tanto no se conseguira ningun retorno de simplemente ejecutarlo, sino que deberemos acceder de alguna forma a este para utilizarlo.
- 'name' le asigna un nombre personalizado al contenedor. Hasta ahora los nombres eran asignados aleatoriamente.
- 'e' configura la variable que contiene la contraseña de la base de datos.

Podemos ver que en la ultima linea nos devuelve una combinacion de numeros y letras que corresponde al hash del container, pero aparte de eso, no imprime nada.

Para ver que esta funcionando podemos correr un comando 'docker container ls'. Lo siguiente es correr algun comando sobre este contenedor:

```
docker exec -it mydb \  
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

En este comando vemos que se usa `exec` en vez de `container`, y lo que esto indica es que `exec` va a correr el comando que se le pase sobre un servidor que ya esta activo (a diferencia de `run` que levanta el container a la vez).

Este comando nos va a mostrar la version de MySQL y va a concluir, pero aun asi no va a matar el container.

Podriamos probar hacer de otra manera esto corriendo primero una consola desde el contenedor y luego ejecutando el mismo comando de antes. Para eso usamos:

```
docker exec -it mydb sh
```

Ahora estamos en una consola dentro del contenedor, por lo cual podemos correr:

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

Y vemos que da el mismo resultado.. Por ultimo para cerrar la sesion de consola corremos: `exit`

1.4 Construir una imagen y correr un container personalizado

En esta actividad vamos a construir una imagen de Docker usando un Dockerfile (que no es mas que un archivo de configuracion parecido a `composer`) y luego compilar esta imagen para levantar un container que contenga esta imagen.

Para eso vamos a entrar en la carpeta de `proyecto1` para ver el Dockerfile que armamos como ejemplo para esta actividad.

Al ver el Dockerfile, podemos encontrar:

- **FROM:** especifica la imagen base que vamos a utilizar para el contenedor, que puede ser desde un `MySQLServer` hasta una instancia de linux en la que querramos correr comandos.
- **COPY:** que va a copiar los archivos presentes en el directorio dentro del container para asi utilizarlos durante la ejecucion de tareas. Copy espera primero la direccion de un archivo presente en la computadora y luego la direccion donde va a copiarlo dentro del container.
- **EXPOSE:** documenta los puertos que la aplicacion va a necesitar. Mas adelante vamos a ver de que manera se pueden vincular estos puertos a puertos reales de nuestra computadora.
- **CMD** especifica que comando correr cuando el contenedor se levanta desde la imagen. Esta opcion se utilizar para configurar el contenedor antes de ingresar en el.

Luego, solo nos queda compilar la imagen para luego ejecutarla.. Para eso vamos a correr el comando:

```
docker image build --tag mi_app:1.0 .
```

En este comando se utiliza el gestor de imagenes para construir una nueva imagen a la cual le otorgamos un nombre personalizado con la opcion `-tag` y le decimos que incluya el directorio actual con el `.`

Ademas al nombre de la imagen le agregamos un numero de version luego de los dos puntos para generar una version nueva mas adelante.

Despues de correrlo vemos que nos imprime todos los pasos que va siguiendo y por ultimo nos indica que la imagen se construyo correctamente.

Ahora podemos correr un `'docker container run'` igual que antes usando la tag que le asignamos previamente a la imagen para correr este container personalizado.. Corremos:

```
docker container run \
--detach \
--publish 80:80 \
--name mi_container \
mi_app:1.0
```

Las novedades de este comando incluyen la opcion `-publish` que como se habia comentado antes, va a servir para vincular puertos reales de nuestra computadora con puertos publicados del contenedor y asi realizar pedidos al contenedor desde fuera.

Ahora podemos ver que nuestro contenedor esta funcionando escribiendo la direccion `'localhost:80'` en un navegador y ver que se despliega tanto la pagina como el servidor.

Ahora que vimos que esta funcionando vamos a eliminarlo para pasar a una version mas dinamica del container.. Vamos a correr:

```
docker container rm --force mi_app:1.0
```

1.5 Modificar un contenedor que esta en ejecución

Al desarrollar, no es conveniente parar y volver a iniciar el contenedor cada vez que se realizan cambios. En eso esta centrado el siguiente ejercicio, donde vamos a montar el directorio con el codigo fuente dentro del contenedor. De esta forma, cualquier cambio efectuado en el directorio de nuestra computadora se va a reflejar directamente sobre el container, cambiando nuestra aplicacion automaticamente.

Para eso vamos a utilizar una `'bind mount'` y para correr nuestro contenedor incluyendo esta opcion vamos a incluir la bandera `-mount all` comando `container run` de antes:

```
docker container run \  
--detach \  
--publish 80:80 \  
--name mi_container \  
--mount type=bind,source="$(pwd)";target=/usr/share/nginx/html \  
mi_app:1.0
```

La bandera mount incluye varias variables que indican:

- type le indica el tipo de montura que va a utilizar; en este caso una bind mount
- source le indica que carpeta quiere montar
- target indica donde va a montar la carpeta, con una direccion dentro del container

Al establecer un bind mount ahora podemos cambiar el contenido de nuestro archivo index.html y ver los cambios reflejados en la pagina, accediendo nuevamente a la direccion en la que el servidor esta corriendo.

Para hacer esta parte mas veloz ya se provee un archivo index-new.html con un cambio de color para la pagina el cual vamos a copiar con el nombre index.html; reescribiendolo para ver los cambios en la pagina.

Vamos a correr:

```
cp index-new.html index.html
```

Y ahora si vamos nuevamente a la direccion del localhost junto con el puerto del container, o recargamos la pagina en caso de que no la hayamos cerrado veremos los cambios.

Ahora que hemos probado las características de los bind mount podemos probar a correr nuevamente el container sin el bind mount para ver si se mantienen los cambios.

Vamos a ejecutar los siguientes comandos:

```
docker rm --force mi_container
```

```
docker container run \  
--detach \  
--publish 80:80 \  
--name mi_container \  
mi_app:1.0
```

Y ahora si probamos a recargar la pagina veremos que esta devuelta en su estado original. Esto es debido a que los bind mount no afectan a la imagen en si sino al

contenido del container y cada vez que se genere un nuevo container con la imagen nuestra va a tener el aspecto original que le otorgamos.

Nuevamente, vamos a detener el container para el siguiente paso:

```
docker rm --force mi_container
```

1.6 Actualizar la imagen de docker

Para hacer persistentes los cambios efectuados localmente en una imagen tendremos que generar una nueva version de la imagen.

Para eso podemos reutilizar el Dockerfile con el que construimos la version anterior, la unica diferencia es que ahora va a copiar el index.html cambiado.

Vamos a correr:

```
docker image build --tag mi_app:2.0 .
```

Y ya tenemos nuestra nueva version de la imagen. Podemos revisar todas las imagenes en el sistema hasta ahora corriendo:

```
docker image ls
```

Ahora podemos probar la nueva version de nuestra aplicacion con un comando anterior:

```
docker container run \
--detach \
--publish 80:80 \
--name mi_acontainer \
mi_app:2.0
```

Y recargando la pagina podremos ver la nueva version de nuestra pagina corriendo desde el container sin bind mounts.

Ademas, podemos poner en simultaneo la otra version de la pagina corriendo, teniendo como precaucion no seleccionar el mismo puerto para los dos containers.

Vamos a correr:

```
docker container run \
--detach \
--publish 8080:80 \
```

```
--name mi_container_antiguo \  
mi_app:1.0
```

Ademas de asignar el puerto 8080 de nuestra computadora para acceder, hay que asignarle un nombre distinto a nuestro nuevo container para que docker los pueda diferenciar.

Ahora accediendo a localhost:8080 podemos ver la version antigua de nuestra aplicacion.

2 Laboratorio de Kubernetes