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 nuestro propia imagen con el codigo 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 distribucion de linux para ver los contenidos de la raiz en el container.

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

Ese ultimo 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 instruccion podemos especificar una subinstruccion para administrarlos que es 'run'.

'run' basicamente corre una instruccion en un nuevo container. De esta forma 'run' simultaneamente crea un nuevo contenedor con la imagen que le pasamos como parametro 'alpine' y corre una instruccion que le pasamos como segundo parametro, en este caso 'ls'.

Luego de correr este comando vamos a ver que docker bajo la imagen oficial desde la pagina y que creo el container para por ultimo correr el comando ls y mostrar la estructura del directorio en la raiz 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 esta utilizando. Ademas, 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
- ity 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 dependecias 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 desplego 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 caracteristicas 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 devido 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

Pagina con entorno de Kubernetes donde ejecutar comandos:
`https://www.katacoda.com/courses/kubernetes/playground`

En kubernetes hay varios conceptos básicos como son los nodos y los pods de los que hablaremos aquí.

Los nodos son computadoras o servidores que se unen a un cluster de nodos para repartirse las tareas de mantenimiento y ejecución de kubernetes.

Recordemos que una de las mejores características de kubernetes es su capacidad de dividir el trabajo y balancearlo entre varias terminales para soportar cargas de trabajo extensivas.

Dentro de los nodos podemos destacar uno especial, que es el nodo master..

Este nodo tendrá una colección de servicios como: - Un servidor API, que nos permite conectarnos con el resto de nuestra aplicación - Servicios principales como el scheduler y el controller manager - etcd que es la base de datos interna para toda la aplicacion

Esto lo hace un nodo principal para el funcionamiento de nuestra aplicación.

Para ejecutar un nodo master luego de instalar correctamente kubernetes podemos correr:

```
kubeadm init
```

Este comando va a mostrar un monton de actividad y en la ultima linea va a tirar algo como:

```
kubeadm join --token SOMETOKEN SOMEIPADDRESS --discovery-token-ca-cert-hash
SOMESHAHAHAHASH
```

Ahora tenemos un cluster corriendo en la red local y cualquier computadora de nuestra red puede conectarse con el script anterior que nos devolvio `kubeadm init` al cluster y dividirse tareas con el nodo master.

Vamos a saltearnos el paso de añadir otro nodo al cluster y vamos directamente al ultimo paso que es configurar el networking para nuestro cluster. Para eso corremos:

```
kubectl apply -n kube-system -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl
```

```
version |base64 |tr -d '\n')"
```

Y asi configuramos el cluster para ahora seguir con la generacion de algunos pods.. Los pods, que aun no se han explicado, son una abstraccion de los containers que nos facilita el trabajo con kubernetes al manejar los containers y la comunicacion entre ellos internamente.

Para empezar deberíamos introducir kubectl que es nuestra herramienta de comunicarnos con kubernetes a través de la consola.

Podemos correr un comando de prueba para ver todos los pods corriendo en el nodo master ahora mismo con:

```
kubectl -n kube-system get pods
```

La opcion -n es para especificar un namespace; en este caso el namespace que kubernetes usa para declarar sus pods internos.

Podemos ver entonces que nos muestra pods como etcd, kube-apiserver, kube-scheduler, etc. de los que ya hemos hablado.

2.1 Corriendo nuestro primer container con Kubernetes:

Podemos iniciar un pod muy sencillo con 'kubectl run' y especificarle una imagen tal como haciamos con docker. Asi, kubernetes va a descargar y utilizar configuraciones default para iniciar nuestra aplicacion corriendo un unico pod.

Vamos a correr:

```
kubectl run pingpong --image alpine ping 8.8.8.8
```

Y entonces?..

Podemos ver que hemos creado unos cuantos servicios nuevos con esos comandos corriendo:

```
kubectl get all
```

Ahi vamos a poder ver los siguientes servicios:

- deploy/pingpong (un deployment que tomo el nombre que especificamos justo luego del run)
- rs/pingpong (un replica set creado por el deployment)
- po/pingpong (un pod creado por el replica set)

Ahora, hemos dado un paso adelante para poder ver la jerarquia existente dentro de kubernetes pero nos hemos adelantado tanto que tocamos conceptos desconocidos como deployment y los replica set.

El concepto de replica set es un concepto diseñado para la utilidad de escalabilidad y confiabilidad que nos ofrece kubernetes. La idea de los replica set es mantener un numero minimo y deseado de pods identicos corriendo para asegurarnos que:

- En caso de que uno o mas pods fallen y desaparezcan, el controlador del replica set va a generar nuevas copias para remplazarlos.
- En el caso de que querramos escalar la aplicación para recibir mas consultas podemos especificar un mayor numero de replicas o configurarlo para que aumente automáticamente.

El concepto de deployment es el ultimo en la jerarquía de kubernetes y actúa orquestando los replica set y los pods con actualizaciones declarativas sobre estos.

Una actualización declarativa es algo como 'podría usted mantener 4 replica set ahora mismo'. De esta forma el deployment puede guiar el estado actual de nuestro proyecto hasta el estado deseado de una manera controlada.

Entonces al configurar nosotros nuestro deployment establecemos un estado deseado y el deployment va a hacer todo lo posible por satisfacer esto utilizando una combinacion de controladores.

Volviendo a lo que íbamos, ahora creamos un container dentro de un pod adentro de muchas otras cosas y esencialmente este container le hizo un ping a la dirección publica de google.

Ahora, todo esto no se entiende hasta que no lo vemos en acción, asi que para eso vamos a correr un nuevo comando:

```
kubectl logs
```

Este comando nos va a mostrar el historial de consola de un pod especifico de un pod si lo pasamos por su nombre; o como alternativa podemos especificar un servicio como los que nos había impreso antes. Como ejemplo vamos a hacer para el deploy:

```
kubectl logs deploy/pingpong
```

Vemos que el nombre tiene que ser el formato que uso al imprimirlo en el paso anterior. Ademas vemos que ahora nos imprimió todos los resultados del ping a google como normalmente.

Nuestro container esta funcionando!!!

Ahora vamos a comprobar el funcionamiento de nuestro replica set. Para eso vamos a eliminar el pod que esta corriendo actualmente y despues ver que automáticamente es reemplazado. Para eso corremos:

```
kubectl get pods
```

Entonces vemos nuestro pod con su nombre y lo usamos en el siguiente comando:

```
kubectl delete pod PODNAME
```

Eso va a eliminar nuestro pod. Mientras lo elimina podemos darle a ctrl + c para seguir usando la consola y correr nuevamente '`kubectl get pods`' para así ver que nuestro pod esta en esta el estado "terminating" o no está, y en su lugar se ha creado un nuevo pod para reemplazarlo.

Por ultimo, vamos a comprobar las capacidades del deployment haciendo que escale la cantidad de replicas requeridas. Para eso vamos a correr:

```
kubectl scale deploy/pingpong --replicas 8
```

El comando se explica por si solo. La única consideración a tener en cuenta es que tenemos que usar el mismo nombre que nos tiro al correr '`kubectl get all`' y va a funcionar.

Ahora si corremos nuevamente '`kubectl get pods`' vamos a ver que ahora el deployment genero de forma automática 8 pods.

Por último podemos eliminar todo el deployment corriendo un:

```
kubectl delete deploy/pingpong
```

2.2 Ejercicio

Como ejercicio final se propone correr la imagen creada en la parte de docker, desde kubernetes con el comando '`kubectl run`' que se explicó y escalarlo 3 veces.