Soluciones al Examen de Prácticas de TSR (Práctica 3) (Tipo A)

(En el test la opción correcta aparece como texto blanco sobre fondo negro. Las opciones correctas son dcba. Para el tipo B se ha cambiado el orden de numeración, siendo correcta la combinación abcd)

Esta prueba incluye 1 pregunta de desarrollo (al final) valorada con 6 puntos, y 4 preguntas de tipo test, valoradas con 1 punto por acierto y -0.33 puntos por error.
Hay que marcar la opción seleccionada en esta misma hoja; para la pregunta de desarrollo se empleará una hoja aparte.

1. Se pretende desplegar un componente web utilizando un contenedor Docker. El código del componente se halla en el directorio **/usr/aplicacion/componentes/web.** También se encuentra en dicho directorio el Dockerfile con el que se creará la imagen del componente. En el Dockerfile del componente web se utiliza el comando **EXPOSE 8000** para indicar que el servidor será accesible a través del puerto 8000 del contenedor.

Indica qué secuencia de comandos utilizarías para realizar dicho despliegue si suponemos que estamos situados en el directorio **/usr/aplicacion/componentes,** y queremos acceder al componente a través del puerto 80 del anfitrión.

a)
```
$ cd web
$ docker build
$ docker run  -p  80:8000 web_image
```
b)
```
$ cd  web
$ docker build -t web_image .
$ docker run  -p  8000:80 image
```
c)
```
$ cd  web
$ docker build -t web_image .
$ docker run  -port 8000 web_image
```
d)
```
$ cd web
$ docker build -t  web_image  .
$ docker run  -p  80:8000 web_image
```

2. **El componente web de la pregunta anterior está en ejecución!!** dentro de un contenedor con identificador **"ef5jf678rlg….".** Este se ha construido a partir de la imagen **web_image.**
*******Hemos realizado una modificación en el código de dicho componente web y pretendemos crear una nueva imagen **web_image.**

Suponemos que estamos situados en el directorio /**usr/aplicacion/componentes/web** (recordamos que el Dockerfile del componente se halla en dicho directorio).

Indica una secuencia válida de comandos para lanzar de nuevo el componente modificado asegurándote que el anterior componente esté eliminado:

a)
```
$ docker rm –f ef5j
$ docker build –t web_image .
```

b)

```
$ docker rmi -f web_image
$ docker build –t web_image .
$ docker run ……
```

c)

```
$ docker rm –f ef5j
$ docker rmi -f web_image
$ docker build –t web_image .
$ docker run ……
```

d)

```
$ docker rm -f web_image
$ docker build –t web_image .
$ docker run ……
```

3. La imagen del componente web con el que estamos trabajando contiene node, la librería binaria de ØMQ y también npm. Sin embargo no tiene instalado el binding para node de zmq. Pretendemos construir una nueva imagen del componente que permita utilizar la librería ØMQ desde node.

Indica cuál de las siguientes afirmaciones es correcta:

a) La única forma de hacerlo es modificando el Dockerfile del componente y volviendo a crear la imagen del componente con "**docker build** …"

b) Se podría ejecutar el contenedor en modo interactivo con "**docker run -i -t ……"** e instalar zmq con "**npm install zmq -g"** directamente en la máquina virtual docker. Posteriormente se podría obtener la nueva imagen con "**docker commit …"** a partir del contenedor previamente parado.

c) Se podría ejecutar el contenedor en modo interactivo con "**docker run -i -t ….."** e instalar zmq con "**npm install zmq -g"** directamente en la máquina virtual docker. La imagen quedaría actualizada automáticamente.

d) Si utilizamos un Dockerfile para obtener la nueva imagen, éste debería contener el comando **"npm install zmq -g".**

4. Ahora le añadimos al componente web un componente de base de datos. Pretendemos utilizar docker-compose para desplegar el nuevo servicio. El fichero **docker-compose.yml** se halla en el directorio **/usr/aplicacion/servicio.** Suponemos que estamos situados en este directorio del anfitrión.

Pretendemos:

1. Lanzar el servicio con todos los componentes.
2. Comprobar que el servicio está en marcha.
3. Parar el servicio con todos los componentes.
4. Eliminar todos los contenedores asociados al servicio.

Indica cuál de las siguientes afirmaciones es cierta:

a) La siguiente secuencia permite realizar las cuatro acciones especificadas:

```
$ docker-compose up
$ docker ps -a
```

```
$ docker-compose stop
$ docker-compose rm -f
```

b) La siguiente secuencia permite realizar las cuatro acciones especificadas:

```
$ docker-compose up
$ docker ps -a
$ docker stop
$ docker rm -f
```

c) La forma habitual de eliminar los contenedores asociados al servicio es borrando, individualmente, cada uno de ellos.

d) Para realizar el despliegue con **"docker-compose up"** es necesario que estén creadas todas las imágenes de los componentes previamente.

5. **PREGUNTA DE DESARROLLO[1].** En el segundo caso de la PARTE 1 de la práctica 3, se describe, configura y despliega un servicio **WordPress**. El fichero **Dockerfile** es:

```
FROM orchardup/php5
ADD . /code
```

… y el **docker-compose.yml** contiene:

```
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    "8000:8000"
  links:
    db
  volumes:
    .:/code
db:
  image: orchardup/mysql
environment:
  MYSQL_DATABASE: wordpress
```

[1] El enunciado es el mismo para la opción B

Una vez desplegado este servicio, en la práctica se solicita la adaptación e inclusión de un componente adicional procedente del myTCPProxy.js de la práctica 1. Aquí se ofrece como solución el siguiente **Lab3myTcpProxyA.js**

Código (Lab3myTcpProxyA.js)

```
var net = require('net');

var LOCAL_PORT = 80;
var LOCAL_IP = '127.0.0.1';
if (process.argv.length != 4) {
  console.error("The WordPress port and IP should be given as
arguments!!");
  process.exit(1);
}
var REMOTE_PORT = process.argv[2];
var REMOTE_IP = process.argv[3];

var server = net.createServer(function (socket) {
  socket.on('data', function (msg) {
    var serviceSocket = new net.Socket();
    serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
      serviceSocket.write(msg);
    });
    serviceSocket.on('data', function (data) {
      socket.write(data);
    });

  });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " +
LOCAL_PORT);
```

Pero disponer de este código no es suficiente para integrar el nuevo componente en un despliegue del servicio WordPress. El componente adicional debe intermediar en las peticiones de servicio, de manera que actúe como receptor de las mismas y las reenvíe al puerto e IP del componente web de WordPress. Para ello **se pide completar los siguientes aspectos**:

1. Crear el **Dockerfile** necesario para desplegar este componente. Debe reenviar las peticiones a la IP y puerto de WordPress. Como referencia puedes *fijarte* en el siguiente Dockerfile usado para el componente **worker** en el sistema client-broker-worker:

```
FROM tsr/fedora-node-devel
COPY ./worker.js worker.js
# We assume that each worker is linked to the broker
# container. Such broker container should have the
# 'BROKER' name and should be linked to this worker.
CMD node worker $BROKER_PORT_8001_TCP
```

2. Nuevo **docker-compose.yml** en el que se incorpore este componente

Solución:

Es importante observar que la aplicación distribuida ha de mantener las mismas interfaces externas, de manera que el puerto 8000, anteriormente asociado al componente web, ahora deberá vincularse al proxy TCP. Esto NO obliga a cambiar el puerto del componente web, sino su visibilidad (p.ej. mediante `expose`) como acceso al servicio en el fichero de despliegue docker-compose.yml.

1. Solución apartado 1 (`Dockerfile`)

Posee una parte general análoga al ejemplo anterior (worker) sobre la que tendremos que aplicar algunos ajustes:

a) El programa ***js*** a ejecutar, que hay que copiar previamente.

b) Hacer accesible el puerto de servicio al que atenderá este proxy TCP
c) Colocar los parámetros adecuados del componente **web** (puerto e IP) para que sean transmitidos al programa en su invocación.

```
FROM tsr/fedora-node-devel
COPY ./Lab3myTcpProxy.js Lab3myTcpProxy.js
EXPOSE 80
CMD node Lab3myTcpProxy $WEB_PORT_8000_TCP_PORT $WEB_PORT_8000_TCP_ADDR
```

Es importante usar la orden adecuada para crear el componente a partir de este Dockerfile, dado que en ella se establece el nombre del componente. Para ello, en el directorio en que se encuentra tanto el Dockerfile como el archivo Lab3myTcpProxy.js, ejecutamos:

```
docker build -t proxy .
```

2. Solución apartado 2 (`docker-compose.yml`**)**

Necesitamos una sección para el componente proxy, destacando:

a) Asociación con el componente que hemos creado (orden `image`)
b) Acceso a la información creada para el componente `web` en su despliegue (valor `web` para la orden `links`)
c) Registro para usar un puerto (8000) del anfitrión como si se tratara del 80 (variable LOCAL_PORT) del contenedor (valor "8000:80" para la orden `ports`)

Y también necesitamos "ocultar" el puerto 8000 del componente `web` para impedir que colisione con el del proxy, y/o que pueda ser accedido directamente sin pasar por el proxy.

a) Retirar orden `ports` de la sección `web`
b) Colocar valor "8000" para la orden `expose` en la sección `web` (permite que sea utilizado internamente)

```
proxy:
  image: proxy
  links:
    - web
  ports:
    - "8000:80"
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  expose:
    - "8000"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

Si se han seguido los pasos anteriores, y nos colocamos en el directorio que contiene este fichero de configuración, para hacer funcionar la magia y arrancar una instancia de cada uno de los 3 componentes ejecutaremos:

```
docker-compose up
```