Spring Boot MySQL

**YAML sintaxis**

<https://en.wikipedia.org/wiki/YAML#Syntax>

Lo mejor que encontré, de muy lejos, es la Wikipedia. Está todo junto, y resumido.

**Docker spring boot MySQL example | Spring boot MySQL docker**

<https://youtu.be/PAQvxqocb6A>

Explica cómo containerizar en Docker. Todo funciona.

**docker compose example | spring boot & mysql docker-compose**

<https://youtu.be/-ekBqIvAGY4>

Es medio desordenado, y no explica todo, da cosas por sobreentendidas. No me gusta.

**Differences Between Docker Compose and Dockerfile by Example**

<https://youtu.be/JmyAMcKUNYA>

May 19, 2022, 11 minutos

Cameron McKenzie. Muy clara la explicación.

**Docker-compose tutorial**

<https://youtu.be/qH4ZKfwbO8w>

Es claro. Una vez que hicimos los otros, y a la vista de este apunte terminado y funcionando, no agrega nada.

**Declare default environment variables in file**

<https://docs.docker.com/compose/env-file/>

No es buena práctica subir a un server archivos de texto plano que tengan, por ejemplo, passwords sin encriptar. En este artículo se discuten opciones más seguras.

**Spring @Value Annotation**

<https://www.digitalocean.com/community/tutorials/spring-value-annotation>

Este es el mejor que pude encontrar.

**Spring Boot - Using ${} placeholders in Property Files**

<https://www.logicbig.com/tutorials/spring-framework/spring-boot/properties-place-holders.html>

Spring proporciona su propia sustitución de variables en los archivos de propiedades. Solo necesitamos usar ${someProp} en el archivo de propiedades e iniciar la aplicación con 'someProp' en las propiedades del sistema o como argumento de clase principal (o jar) '--someProp=theValue'. Por ejemplo: en la clase principal

Esta característica nos permite usar argumentos de línea de comando 'cortos'.

**Ejemplo**: en el archivo

src/main/resources/application.properties

app.title=Boot ${app} @project.artifactId@

<https://stackoverflow.com/questions/38983934/cannot-get-maven-project-version-property-in-a-spring-application-with-value>

Para obtener acceso a las propiedades de Maven en la aplicación Spring Boot, todo lo que necesitamos es mapearlas con el delimitador @ en application.properties como por ejemplo:

app.version=@project.version@

app.name=@project.name@

Luego, utilícelos en la aplicación como propiedades ordinarias, por ejemplo:

@Service

public class SomeService {

@Value("${app.version}")

private String appVersion;

// other stuff

}

# Arrancar WampServer

Vamos a armar la base de datos en WampServer, porque es muy simple. Antes de arrancar WampServer necesitamos asegurarnos de que el puerto 3306 (el de MySQL) no está en uso. En la consola de Windows:

for /f "tokens=5" %a in ('netstat -aon ^| find /i "LISTENING" ^| find ":3306"') do taskkill /f /pid %a

# Crear la base de datos y el usuario

Una vez en WampServer, arrancar phpMyAdmin. En el archivo

\40-docker\crud\sql\anime.sql

está el script SQL para crear la base de datos y el usuario. Copiar y pegar todo en el panel SQL de phpMyAdmin. Una vez ejecutado, tenemos la base de datos y el usuario.

# Aplicación Spring Boot

El autor usa su propia aplicación. Yo voy a usar la que ya tengo hecha. Está en la carpeta crud. Abrirla con VS Code. Es una aplicación de Spring Boot, con API REST. Usa una base de datos MySQL también propia de él, pero yo sigo con la que ya tenía hecha para mi aplicación.

# Credenciales de la aplicación para la base de datos

Las credenciales y otros parámetros de la configuración necesarios para que la aplicación se conecte a la base de datos están en el archivo:

src\main\resources\application.properties

# Debe ser igual al artifactId del pom

spring.application.name=crud

server.port=8080

spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver

# Esta es para correr en el IDE, fuera de un container

spring.datasource.url=jdbc:mysql://localhost:3306/anime

# Esta es para correr en el container

# spring.datasource.url=jdbc:mysql://mysqldb:3306/anime

spring.datasource.username=cruduser

spring.datasource.password=1234

# https://stackoverflow.com/questions/50322550/hibernate-dialect-for-mysql-8

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

Notar que el host es diferente según MySQL esté corriendo en la PC local por fuera de un container (localhost), o dentro de un container (mysqldb, que es el nombre del container).

Hay algunos mecanismos para cargar parámetros de configuración en run time.[[1]](#footnote-1) Nosotros no los vamos a ver, porque está fuera del alcance de este curso. Simplemente mencionamos que existen, y si alguien necesita profundizar en ese tema, puede comenzar por el artículo citado.

# Ejecutar en la PC local desde el IDE con MySQL en WampServer

Si uno tiene corriendo WampServer y ejecuta la aplicación desde el IDE, se ve que todo funciona correctamente. Hay que asegurarse de que el puerto 3306 no está siendo usado, por ejemplo por un container.

En la URL

<http://localhost:8080/all>

se puede ver la lista de todos los animes cargados en la base de datos. Esto es lo que hemos hecho muchas veces, es conocido y no tiene nada de sorprendente.

Recordar que la aplicación Java se conecta a una base de datos MySQL, y por eso necesitamos que el motor de la base de datos esté corriendo, lo cual hemos hecho hasta ahora usando WampServer.

# Docker Desktop y CLI

Descargar el instalador desde:

<https://www.docker.com/products/docker-desktop/>

Se instala una aplicación de Windows que se llama Docker Desktop. Además se instala un comando, que se puede ejecutar desde la consola de Windows, escribiendo docker. Este comando es un gestor de containers autosuficiente.

Siempre que queremos usar un comando docker en la terminal de Windows, antes debemos asegurarnos de que esté corriendo Docker Desktop.

# Extensión Docker de Microsoft

En Visual Studio Code instalamos esta extensión. Notar que, lo mismo que el intérprete de línea de comando, esta extensión requiere que Docker Desktop esté corriendo.

# Imágenes en Docker

En la extensión de VS Code la solapa Images, o bien en laconsola de Windows con el comando:

docker images

# MySQL: descargar la imagen

<https://youtu.be/PAQvxqocb6A?t=134>

Si no tengo la imagen de MySQL ya descargada en Docker, la tengo que bajar de hub.docker.com. En la consola de Windows:

docker pull mysql

La sintaxis general de este comando[[2]](#footnote-2) es:

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

docker pull: baja a la PC local una imagen o un repositorio desde un registro. Por defecto, descargará las imágenes desde Docker Hub[[3]](#footnote-3).

[OPTIONS] En este caso no tenemos opciones.

mysql es el nombre de la imagen. El formato general del parámetro NAME es:

PATH\_AL\_REGISTRO:PUERTO/NOMBRE\_DE\_LA\_IMAGEN: TAG|@DIGEST]

Por ejemplo:

docker pull myregistry.local:5000/testing/test-image

descarga la imagen llamada testing/test-image desde un registro local por el puerto 5000. El path del registro es similar a una URL, pero no tiene el protocolo. Docker usa por defecto https://. Se puede configurar un registro para que acepte conexiones no seguras, pero nosotros no lo vamos a hacer.

Una vez que termina de descargar la imagen, podemos comprobar en la extensión de Visual Studio Code, en la solapa Images. Deberíamos ver que tiene el tag latest.

# MySQL: crear un container

Ahora que tengo la imagen en mi PC local, tengo que instanciar un container para MySQL. Asegurarse de que el puerto no está en uso. Por ejemplo, hay que apagar WampServer.

Crear un volumen interno de Docker:

docker run -d --name mysqldb -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -e MYSQL\_PASSWORD=1234 -e MYSQL\_USER=cruduser -e MYSQL\_DATABASE=anime -v mi-container-docker:/var/lib/mysql mysql

Crear un volumen en una carpeta física de la PC local:

docker run -d --name mysqldb -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -e MYSQL\_PASSWORD=1234 -e MYSQL\_USER=cruduser -e MYSQL\_DATABASE=anime -v %CD%/mysqldb:/var/lib/mysql mysql

docker run: primero crea una capa de contenedor grabable sobre la imagen especificada y luego la inicia usando el comando especificado.

-d: ejecutar el container en background (detach)

--name mysql-c: asigna el nombre mysqldb al container

-p 3306:3306: publica un puerto del container al host puerto\_local:puerto\_en\_el\_container

-e MYSQL\_ROOT\_PASSWORD=123456: establece la variable de entorno

-v volumen\_local:/var/lib/mysql crea un volumen, es decir que la base de datos va a estar guardada en una carpeta local y no se va a perder si borramos el container. Si la carpeta final (o sea la última parte del path) no existe, la crea. Pero todo lo demás debe existir.

mysql: nombre de la imagen a ejecutar.

Notar que, además de la password de root, estamos pasando el nombre de una base de datos, el nombre de un usuario que no es root, y la password de ese usuario. Esta información estará en el volumen, y esto nos permitirá conectarnos desde MySQL WorkBench, por ejemplo, a la base de datos anime con el usuario cruduser. Esto no lo podríamos hacer si no pusiéramos los parámetros antes discutidos. Por supuesto que siempre podríamos conectarnos como root, pero eso no es seguro.

Ahora tengo MySQL server corriendo en un container, con las credenciales que elegí.

Podemos comprobar en la extensión de Visual Studio Code, en la solapa Containers.

Si voy a <http://localhost:3306/> en realidad no voy a ver nada. El server está listening, pero esa URL no le está mandando ningún request. Podría usar el phpMyAdmin, pero solo lo tengo en el WampServer, así que no puedo. Entonces uso una alternativa.

Ese container tiene un puerto interno, que es por default el 3306. Por el modo en que lo construí, el puerto interno del container está mapeado al puerto 3306 de mi localhost. De modo que yo me puedo conectar de dos formas diferentes. Desde dentro de una red en la que esté el container, el host es el nombre del container. Desde fuera de una red de containers, el host es por supuesto el localhost.

# MySQL Workbench 8.0 CE

Descargo MySQL Workbench Community Edition de:

<https://dev.mysql.com/downloads/workbench/>

Lo instalo y lo ejecuto. Me conecto al servicio de MySQL que está corriendo en el container. Notar que no está corriendo en el localhost. El host es el container. y está listening en el puerto 3306, porque así lo configuré. Y como esto es lo que asume MySQL Workbench por defecto, no hay problemas. Uso las credenciales que establecí al crear el container. En el file system del container, MySQL guarda las bases de datos en la carpeta /var/lib/mysql dentro del file system del container.

# File system del container

El container tiene un file system bastante complejo.

Visual Studio Code tiene una extensión para Docker. Si la tenemos instalada, nos permite examinar todo con gran detalle. Es muy intuitiva y poderosa. Podemos comprobar que las bases de datos están donde dijimos antes.

Otro modo de explorarlo es desde Docker Desktop. Vamos a la solapa containers. Veremos la lista de los containers que tenemos. En el costado derecho hay varios botones o iconos, que nos permiten arrancar, pausar, reiniciar, detener o borrar cada container. Entre estos botones, los dos de la izquierda nos permiten ver el container en el browser, o bien en una terminal. Justamente en la terminal tipo Linux podremos ver el file system.

# \40-docker\crud\sql\anime.sql

En este archivo está el script SQL para crear la base de datos y el usuario. La base de datos que creamos antes con este mismo script está en el sistema de archivos de WampServer, en el lugar donde MySQL guarda las bases de datos. El MySQL que está ahora corriendo en el container está limpio, no tiene ninguna base de datos. Copiar y pegar todo en el panel Query 1 del Workbench. Una vez ejecutado, refrescamos el panel de bases de datos, y vemos que ya tenemos la base de datos y el usuario, todo en el container.

Notar que el container que creamos hace un momento tiene ahora una base de datos, y un usuario que no es el root, con una password diferente.

Notar que estas credenciales deben ser exactamente iguales a las que están en el archivo application.properties, que es de donde las tomará el jar cuando vayamos a ejecutarlo.

Este script solo tendremos que ejecutarlo la primera vez que arranquemos el container. Como ese container tiene asociado un volumen local, la base de datos que se crea queda grabada en esa carpeta local. Si borramos el container la base de datos no se pierde. Al crear o reiniciar el container, los datos están.

# Ejecutar en la PC local desde el IDE con MySQL en container

Arranco la aplicación desde el IDE, como antes. En la URL

<http://localhost:8080/all>

se puede ver la lista de todos los animes cargados en la base de datos. La aplicación está ahora conectada a la base de datos que está en el container de MySQL. Lo que ahora sucede es que en el puerto 8080 está listening la aplicación Java. A esa aplicación Java le mando un request. La aplicación Java lo recibe, y como tiene un controlador REST le manda a su vez un request al puerto 3306, que es donde está listening el servicio MySQL. Y por eso me llega la respuesta. Notar que ahora no me vino desde el MySQL que está dentro de WampServer, sino desde el MySQL que está corriendo en el container que arranqué desde Docker.

# Detener el IDE

Detener la aplicación que está corriendo en el IDE. Comprobar que la URL <http://localhost:8080/all> no tiene la aplicación.

# Crear network en Docker

Cada container que se crea como una instancia de una imagen corre aislado, solo se ve a sí mismo. Para que dos containers puedan verse entre sí, necesitamos tener una network, y que ambos containers estén en ella.

docker network create crud-n

Crea una network y le llama crud-n.

docker network connect crud-n mysqldb

Conecta a la network antes creada el container de la base de datos, que hasta ahora estaba aislado. Comprobamos en la extensión Docker. Si no aparece, refrescamos la red y aparecerá. Ahora el container de la base de datos está en la red, pero está solo.

# Credenciales de la aplicación para la base de datos

Editar el archivo:

src\main\resources\application.properties

# Debe ser igual al artifactId del pom

spring.application.name=crud

server.port=8080

spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver

# Esta es para correr en el IDE, fuera de un container

# spring.datasource.url=jdbc:mysql://localhost:3306/anime

# Esta es para correr en el container

spring.datasource.url=jdbc:mysql://mysqldb:3306/anime

spring.datasource.username=cruduser

spring.datasource.password=1234

# https://stackoverflow.com/questions/50322550/hibernate-dialect-for-mysql-8

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

Notar que el host es diferente según MySQL esté corriendo en la PC local por fuera de un container (localhost), o dentro de un container (mysql-c, que es el nombre del container). Recompilar la aplicación Spring Boot.

# Dokerfile para construir la imagen de Spring Boot

Ya tenemos dockerizado el server de MySQL. Ahora tenemos que dockerizar la aplicación de Spring Boot.

Docker puede crear imágenes automáticamente leyendo las instrucciones de un Dockerfile[[4]](#footnote-4). Un Dockerfile es un documento de texto que contiene todos los comandos que un usuario podría llamar en la línea de comandos para ensamblar una imagen. Usando docker build se puede crear una construcción automatizada que ejecuta varias instrucciones de la línea de comandos en sucesión. Para eso, se usa el comando docker build. Pero para poder usarlo, antes hay que escribir el Dockerfile.

Vamos a la carpeta de la aplicación, 40-docker\crud, creamos un archivo de texto y lo llamamos Dockerfile, sin extensión. En ese archivo pegamos lo siguiente:

# eclipse-temurin soporta versiones recientes del JDK.

# En muchos tutoriales usan openjdk, pero la documentación

# oficial dice que está deprecado.

FROM eclipse-temurin

ADD target/\*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

La instrucción FROM[[5]](#footnote-5) inicializa una nueva etapa de la construcción y establece la imagen base para las instrucciones posteriores. En este caso, lo que estamos diciendo es que queremos arrancar la construcción con eclipse-temurin[[6]](#footnote-6) que es una alternativa a openjdk.

La instrucción ADD[[7]](#footnote-7) copia nuevos archivos, directorios o direcciones URL de archivos remotos desde <src> y los agrega al sistema de archivos de la imagen en la ruta <dest>. En este caso, le estamos diciendo que copie todos los archivos de tipo jar que hay en la carpeta target (<src>, relativo a la carpeta en la que está el archivo Dockerfile) en la carpeta app.jar (<dest> es el path en el file system de la imagen).

Para construir la imagen de la aplicación Spring Boot, arrancar una consola de Windows en la carpeta en la que está el Dockerfile (que en este caso debería ser 40-docker\crud) y pegar el comando:

docker build -t crud .

El comando docker build[[8]](#footnote-8) construye una imagen Docker según las instrucciones del archivo Dockerfile. El parámetro -t sirve para darle un nombre a la imagen. El dot que está al final es el path, en este caso relativo a la carpeta en la que está físicamente el archivo Dockerfile. Ese path determina cuál es el contexto sobre el que va a trabajar Docker.

En Visual Studio Code vamos a la extensión de Docker, solapa Images, y vemos que hay una imagen llamada crud, con el tag latest.

# Instanciar un container a partir de la imagen

Antes de ejecutar este comando, hay que asegurarse de que el puerto 8080 está libre. Entre otras cosas, si la aplicación está corriendo en el IDE, hay que apagarla. Si nos olvidamos, y nos da un error, el container de todos modos fue construido. En ese caso, habría que borrarlo y recomenzar.

Puede dar el error de que la app fue compilada con una versión de Java posterior a la del container. Hay que recompilar la app.

Para instanciar un container a partir de esta imagen y conectarlo a la network:

docker run --name crudapp --net crud-n -p 8080:8080 -e MYSQL\_HOST=mysqldb -e MYSQL\_USER=cruduser -e MYSQL\_PASSWORD=1234 -e MYSQL\_PORT=3306 crud

docker run: primero crea una capa de contenedor grabable sobre la imagen especificada y luego la inicia usando el comando especificado.

--name crudapp: asigna el nombre al container.

--net crud-n: conecta este container a la network que antes creamos.

-p 8080:8080: publica un puerto del container al host.

-e MYSQL\_HOST= mysqldb: establece la variable de entorno .

-d: ejecutar el container en background (detach).

crud: nombre de la imagen a ejecutar.

Notar que las credenciales que le damos ahora son solo para la base de datos de la aplicación. No le pasamos la password del root. Esa solo se necesita para arrancar mysql, pero ya está arrancado.

# Ejecutando todo en container

Esta vez la aplicación arranca automáticamente. En la URL

<http://localhost:8080/all>

se puede ver la lista de todos los animes cargados en la base de datos. La aplicación está ahora conectada a la base de datos que está en el container de MySQL. Lo que ahora sucede es que en el puerto 8080 está listening la aplicación Java. A esa aplicación Java le mando un request. La aplicación lo recibe, y como tiene un controlador REST le manda a su vez un request al host del container, no mi localhost. Lo manda al puerto 3306. que es donde está listening el servicio MySQL. Y por eso me llega la respuesta. Notar que ahora no me vino desde el MySQL que está dentro de WampServer, sino desde el MySQL que está corriendo en el container que arranqué desde Docker.

# Diferencia entre Docker Compose y Dockerfile

Hasta ahora lo que hemos visto es cómo se usa Dockerfile para construir imágenes de Docker, cómo se crea una network y cómo se conectan dos containers a la misma network para que se vean entre sí.

En pocas palabras:

* Dockerfile es solo para construir la imagen, pero no para correr containers.
* docker-compose up es para correr containers; construye imágenes con la opción build.

Si docker-compose necesita una imagen que no tiene disponible, va a tratar de buscarla en el registro default. Si la encuentra la descarga. Si no la encuentra da un error fatal.

# Imagen y container de nginx

En la consola de Windows pegamos:

docker pull nginx

Si ya tenemos descargada nginx:latest nos lo informa. En caso contrario la descarga desde el registro default. Arrancamos un container con esta imagen, por ejemplo desde la extensión de VS Code. En la solapa Containers de la extensión de VS Code veremos que aparece un container basado en la imagen nginx:latest, con un nombre aleatorio, porque no le dimos uno. Navegamos por el file system de este container, hasta la carpeta \usr\share\nginx\html\. Veremos que hay dos archivos: index.html y 50x.html. Si abrimos este container en el browser, veremos justamente el contenido de index.html. Si nosotros ponemos en esta carpeta todo nuestro sitio web, lo veremos en el browser.

# Dockerfile con nginx y website

Vamos a la carpeta MYWEBSITE. En esa carpeta creamos un archivo Dockerfile y en ese archivo pegamos las siguientes líneas:

FROM nginx

COPY ./website /usr/share/nginx/html/

Como ya vimos, la primera línea establece la imagen que se tomará como base para todo lo que sigue.

La segunda instrucción[[9]](#footnote-9) copia archivos o directorios desde un origen a un destino. En este caso, copia toda la carpeta website (que contiene el web site) al lugar que antes discutimos dentro del file system del container. Tenemos que reconocer que el website es francamente humilde: tiene un archivo html, un js y un css. Pero es suficiente para mostrar lo que queremos.

Abrimos una terminal de Windows en la carpeta MYWEBSITE y pegamos el comando:

docker build -t nginx-i .

Como ya hemos visto, este comando construirá una imagen siguiendo las instrucciones del archivo Dockerfile, cuyo contexto será el directorio en el que estamos, y le pondrá como nombre nginx-i. Podemos verficar esto en VS Code. Si no lo vemos en Docker Desktop, tenemos que ir a Troubleshoot y reniniciar el daemon.

Ya tenemos la imagen construida, pero todavía no hay ningún container.

# Compose file

El nombre preferido para este archivo, según la documentación oficial, es compose.yaml. El formato del archivo compose.yaml está definido en:

<https://github.com/compose-spec/compose-spec/blob/master/spec.md#compose-file>

La sintaxis[[10]](#footnote-10) es simple. Lo único que vamos a decir por ahora es que la indentación es muy importante, porque se requiere para dar estructura al contenido. Solo se usan espacios, no tabulaciones, y cada nivel está indentado exactamente dos espacios.

# Version top-level element

<https://github.com/compose-spec/compose-spec/blob/master/spec.md#version-top-level-element>

Muchos tutoriales y ejemplos ponen al principio del archivo el parámetro version. Pero la documentación oficial dice que está deprecado. Yo no lo voy a usar.

# Docker compose

En lugar de correr los containers con docker run, vamos a usar Docker Compose. Para eso, navegamos a la carpeta MYWEBSITE. En esa carpeta creamos un archivo compose.yaml y en ese archivo pegamos las siguientes líneas:

services:

  nginx-s:

    container\_name: website-c

    image: nginx-i

    cpus: 1.5

    mem\_limit: 2048m

    ports:

      - "8080:80"

Este archivo establece cómo queremos ejecutar el container a partir de la imagen.

La primera línea dice que vamos a correr servicios. Un servicio es un server corriendo en un puerto, algo que podemos consumir con un cliente.

La segunda línea dice que nuestro servicio se va a llamar nginx-s.

Este servicio va a correr en un container que se va a llamar website-c.

El container va a ser una instancia de la imagen nginx-i, que es la que acabamos de crear en el paso anterior.

Queremos que tenga 1.5 cpus y 2 gigabytes de memoria.

Finalmente mapeamos el puerto 8080 de nuestro localhost al puerto 80 interno del container, que es el puerto default de nginx.

# Correr un container con Docker Compose

En la consola de Windows pegamos el comando:

docker-compose up

Si vamos a <http://localhost:8080/> veremos nuestro sitio web.

# Correr varios servicios con docker-compose

Para empezar apagamos todos los containers anteriores. Inmediatamente comprobaremos que el sitio no se ve más en el browser.

Navegamos a la carpeta 40-docker\MYOTHERWEBSITE. Creamos el archivo compose.yaml. Agregamos al archivo compose.yaml lo siguiente:

services:

  apache-s:

    container\_name: website-c

    image: httpd:latest

    cpus: 1.5

    mem\_limit: 2048m

    ports:

      - "80:80"

    volumes:

      - ./website:/usr/local/apache2/htdocs

Este servicio está corriendo en un container que está en mi PC local. Este servicio expone un puerto a través de su host. Si quiero acceder a este servicio desde dentro de una red de containers de la que este container sea parte, el host será el nombre del servicio. Si quiero acceder al servicio desde mi PC pero por fuera del container, el host será mi localhost. Si estuviera en la nube, el host tendría su correspondiente URL.

Ejecutamos el comando docker-compose up. Vemos que hay una imagen que antes no estaba, que es hpptd:latest. Se descargó automáticamente del repositorio default. Vemos que hay una nueva network, que se llama myotherwebsite\_default. Dentro de ella hay un nuevo container, que antes no estaba, que es una instancia de la imagen httpd:latest y se llama website-c. Si vamos al file system de este container, y navegamos hasta /usr/local/apache2/htdocs, veremos que allí están copiados los tres archivos de la carpeta website, aunque nosotros no lo hicimos explícitamente. Lo que sucede es que la primera vez que se levana el container se crea el volumen, para lo cual se copia la carpeta del origen al destino. El destino es el punto de arranque default de httpd.

Probamos esto con la URL <http://localhost/>, que usa el puerto 80, y ahora dirá en el título MYOTHERWEBSITE, simplemente para que estemos seguros de que no nos estamos confundiendo con lo que hicimos en los pasos anteriores.

Detenemos todos los containers, borramos la red y el volumen. Vamos a agregar un segundo servicio al archivo compose.yaml:

  nginx-s:

    container\_name: nginx-c

    image: nginx:latest

    cpus: 2.5

    mem\_limit: 4096m

    ports:

      - "99:80"

    volumes:

      - ./nginx-app:/usr/share/nginx/html

Este nuevo servicio se llamará nginx-s, y correrá en un container que se va a llamar nginx-c. Se armará a partir de la imagen nginx:latest, que habíamos bajado antes, pero la borramos después de usarla como base para construir la imagen nginx-i. Tampoco puedo usar nginx-i, porque si bien a esta última la construí a partir de nginx, le copié el otro sitio web al punto de inicio, así que no me va a servir. Por eso arranco de nuevo, y le armo el volumen con el sitio nuevo, que es el que tengo ahora. Vamos a armar el volumen copiando de la carpeta local nginx-app a la carpeta /usr/share/nginx/html del file system del container. Ejecutamos el comando docker-compose up y revisamos. Si ahora comprobamos en la URL <http://localhost:99/>, vemos que allí está nuestra aplicación. En el file system del container navegamos hasta /usr/share/nginx/html, y allí vemos que se copió el archivo index.html.

# Desplegar un war en un container Docker

Seguimos las instrucciones de la documentación oficial:

<https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/>

The example app has been packaged as a war file and can be downloaded [here](https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war) (Note: make sure your browser doesn't change file extension or append a new one).

Descargamos el archivo sample.war y lo guardamos en la carpeta samplewar.

The easiest way to run this application is simply to move the war file to your **CATALINA\_HOME/webapps** directory. Tomcat will automatically expand and deploy the application for you. Nosotros vamos a hacer algo más simple: dejaremos que docker-compose se ocupe de todo.

Agregamos las siguientes líneas a compose.yaml:

  tomcat-s:

    container\_name: samplewar-c

    image: tomcat:latest

    cpus: 3.5

    mem\_limit: 4096m

    ports:

      - "8080:8080"

    volumes:

      - ./samplewar:/usr/local/tomcat/webapps

Como ya sabemos, **CATALINA\_HOME** es la carpeta donde está instalado Tomcat dentro del container Docker. En este caso, /usr/local/tomcat. Esto lo podremos ver una vez que el container arranque, no antes.

Cuando ejecutamos el comando docker-compose up vemos que el war fue expandido en la carpeta local samplewar y se creó una nueva carpeta sample, en la cual ahora podemos ver una estructura de directorios que antes no estaban. Si exploramos el file system del container recién creado, veremos que en la carpeta destino designada ahora está copiado no solo el war, sino la carpeta sample con todo su contenido.

You can view it with the following URL (assuming that you're running tomcat on port 8080 as is the default):

<http://localhost:8080/sample>

# Desplegar un jar en un container Docker

Seguimos los mismos pasos que en la primera parte de este tutorial. Navegamos hasta la carpeta 40-docker\MYOTHERWEBSITE\java. Dentro de la carpeta java hay una aplicación Spring Boot muy simple. Creamos el archivo Dockerfile y pegamos el siguiente texto:

FROM eclipse-temurin

ADD target/\*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

Generamos la imagen de esta aplicación con el comando:

docker build -t jar-i .

Comprobamos que tenemos la imagen jar-i en VS Code.

Agregamos a compose.yaml el texto:

  tomcat-s2:

    container\_name: jar-c

    image: jar-i:latest

    cpus: 3.5

    mem\_limit: 4096m

    ports:

      - "8081:8080"

    volumes:

      - ./samplewar:/usr/local/tomcat/webapps

Volvemos a la carpeta MYOTHERWEBSITE y ejecutamos el comando docker-compose up. Vemos que ahora tenemos cuatro servicios, cada uno de ellos corriendo en su propio container.

<http://localhost/>

<http://localhost:99/>

<http://localhost:8080/sample>

<http://localhost:8081/>

# Volvemos al principio

Detenemos todos los containers, borramos todas las imágenes y los volúmenes, cerramos todas las carpetas, navegamos hasta \40-docker\crud, o sea donde estábamos al principio.

Nos aseguramos de que application.properties quede así:

# Debe ser igual al artifactId del pom

spring.application.name=crud

server.port=8080

spring.datasource.driver-class-name =com.mysql.cj.jdbc.Driver

# Esta es para correr en el IDE, fuera de un container

# spring.datasource.url=jdbc:mysql://localhost:3306/anime

# Esta es para correr en el container

spring.datasource.url=jdbc:mysql://mysqldb:3306/anime

spring.datasource.username=cruduser

spring.datasource.password=1234

# https://stackoverflow.com/questions/50322550/hibernate-dialect-for-mysql-8

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

#spring.jpa.show-sql: true

logging.level.org.hibernate.SQL=debug

Dockerfile debe quedar así:

FROM eclipse-temurin

COPY target/\*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

Creamos un archivo compose.yaml y le pegamos este texto:

## compose.yaml

services:

  # Versión de

  # https://youtu.be/6hMHziv0T2Y

  # Dockerize Spring Boot Application with MySQL using Docker compose

  mysqldb:

    # Solo en Linux. En Windows no va. ¡JA ja ja !

    # platforn: linux/x86\_64

    image: mysql

    restart: always

    ports:

      - 3306:3306

    environment:

      MYSQL\_DATABASE: anime

      MYSQL\_USER: cruduser

      MYSQL\_PASSWORD: 1234

      # Mandatorio, pero puede ser cualquier cosa

      MYSQL\_ROOT\_PASSWORD: 123456

    # Este script se va a ejecutar la primera vez, y es un modo de

    # inicializar la base de datos. Esto lo podría hacer manualmente,

    # pero de este modo es automático.

    volumes:

      - "./sql/anime.sql:/docker-entrypoint-initdb.d/1.sql"

  server:

    build: .

    # Este servicio depende de mysqldb. Si la base de datos no arrancó,

    # no se puede conectar. Pero puede pasar que no haya nada mal,

    # y que solo necesite más tiempo. Por eso le digo que siga intentando

    # arrancar hasta que pueda.

    restart: always

    ports:

      - 8080:8080

    depends\_on:

      # acá va el nombre de un servicio, no de un container

      - mysqldb

Este archivo se ejecuta con docker-compose up.

Notar que se ocupa de buildear la imagen de la aplicación crud, si no existe. Lo que no hará, porque no puede, es compilar y generar el jar. El jar ya debe estar en target. Si está, se construye la imagen. Si no está, hay un error.

1. <https://www.baeldung.com/spring-boot-yaml-vs-properties> [↑](#footnote-ref-1)
2. <https://docs.docker.com/engine/reference/commandline/pull/> [↑](#footnote-ref-2)
3. <https://hub.docker.com/> [↑](#footnote-ref-3)
4. <https://docs.docker.com/engine/reference/builder/> [↑](#footnote-ref-4)
5. <https://docs.docker.com/engine/reference/builder/#from> [↑](#footnote-ref-5)
6. <https://hub.docker.com/_/eclipse-temurin> [↑](#footnote-ref-6)
7. <https://docs.docker.com/engine/reference/builder/#add> [↑](#footnote-ref-7)
8. <https://docs.docker.com/engine/reference/commandline/build/> [↑](#footnote-ref-8)
9. <https://docs.docker.com/engine/reference/builder/#copy> [↑](#footnote-ref-9)
10. <https://en.wikipedia.org/wiki/YAML#Syntax> [↑](#footnote-ref-10)