Docker en Spring Boot Standalone App de Docker

Autor: Alberto Herrera Poza
Github: https://github.com/AlbertoIHP

Introducción

El presente documento tiene la finalidad de levantar un contenedor a partir de una aplicación Java en Spring Boot. Esto con la finalidad de crear un primer archivo Dockerfile, y analizar su sintaxis. Además, poder visualizar un par de comandos relevantes del Core de Docker.

1. Obteniendo la app

Para poder obtener el proyecto, en primera instancia es necesario clonar el siguiente repositorio.

```
git clone https://github.com/callicoder/spring-boot-websocket-chat-demo
cd spring-boot-websocket-chat-demo
touch Dockerfile
```

2. Dockerfile

El Dockerfile, es el archivo configuración base para levantar un contenedor con Docker, este maneja un lenguaje bastante sencillo y expresivo, mediante los cuales, el programador o usuario de Docker, puede definir cuál es el proceso necesario para levantar una aplicación.

```
# Start with a base image containing Java runtime
FROM openjdk:8-jdk-alpine
# Add Maintainer Info
LABEL maintainer="callicoder@gmail.com"
# Add a volume pointing to /tmp
VOLUME /tmp
# Make port 8080 available to the world outside this container
EXPOSE 8080
# The application's jar file
ARG JAR_FILE=target/websocket-demo-0.0.1-SNAPSHOT.jar
# Add the application's jar to the container
ADD ${JAR_FILE} websocket-demo.jar
# Run the jar file
ENTRYPOINT
["java","-Djava.security.egd=file:/dev/./urandom","-jar","/websocket-dem
o.jar"]
```

Los comandos utilizados en este Dockerfile y otros que pueden ser útiles.

- **FROM**: Permite utilizar otra imagen Docker, desde la cual pueda levantarse el nuevo contenedor.
- **LABEL**: Permite agregar metadata a la imagen.
- **VOLUME**: Los volúmenes son un mecanismo para conservar los datos generados por el contenedor en el sistema operativo anfitrión y compartir directorios del sistema operativo anfitrión con el contenedor. (/tmp porque aquí es donde la aplicación Spring Boot crea directorios de trabajo para Tomcat de forma predeterminada.)
- **EXPOSE**: Permite exponer puertos arbitrariamente.
- **ARG**: Permite definir variables de entorno en el contenedor QUE NO PERSISTEN en sub capas.
- **ADD**: Permite copiar archivos al contenedor.
- **ENTRYPOINT**: La instrucción ENTRYPOINT le permite configurar un contenedor que se ejecutará como un ejecutable. Se parece a CMD, porque también le permite especificar un comando con parámetros. La diferencia es que el comando ENTRYPOINT y los parámetros no se ignoran cuando el contenedor Docker se ejecuta con parámetros de línea de comandos. (Hay una forma de ignorar ENTRYPOINT, pero es poco probable que lo haga).
- **WORKDIR:** La instrucción WORKDIR establece el directorio de trabajo para cualquier instrucción RUN, CMD, ENTRYPOINT, COPY y ADD que la siga en el Dockerfile. Si el WORKDIR no existe, se creará incluso si no se usa en ninguna instrucción posterior de Dockerfile.
- **COPY:** La instrucción COPY copia nuevos archivos o directorios de <src> y los agrega al sistema de archivos del contenedor en la ruta <dest>.
- **RUN:**La instrucción RUN ejecutará cualquier comando en una *nueva capa encima de la imagen actual* y confirmará los resultados. La imagen comprometida resultante se usará para el siguiente paso en el Dockerfile.
- **ENV:**La instrucción ENV establece la variable de entorno <key> en el valor. Este valor estará en el entorno de todos los comandos Dockerfile "descendientes" y también se puede reemplazar en línea en muchos.
- **CMD**: Solo puede haber una instrucción CMD en un Dockerfile. Si enumera más de una CMD, solo la última CMD surtirá efecto. Si el contenedor se ejecuta con un comando, CMD será ignorado.

3. *Dockerfile build*

El **núcleo de Docker,** permite a raíz de un **Dockerfile**, mediante el comando **build**, crear un **contenedor** reflejado en su propia **imagen** (Del mismo tipo que ocupamos en el Dockerfile con el comando **FROM**). Esta imagen, una vez creada, podrá ser levantada mediante el comando **run** del núcleo de Docker.

Para compilar el proyecto Java, es necesario correr el siguiente comando:

```
mvn clean package
```

Con esto, tendremos un fichero con extensión JAR, y es posible entonces compilar la imagen del contenedor en cuestión.

```
docker build -t spring-boot-websocket-chat-demo .
```

4. *Dockerfile run (And other useful commands)*

Una vez compilada nuestra imagen de manera exitosa, podemos visualizar el total de imágenes instaladas en nuestro núcleo de Docker.

```
docker image ls
```

Para correr la imagen compilada, se puede utilizar el comando **run**, el que dentro de otros parámetros o flags, tienen el selector de puerto (-p), donde usa la sintaxis **host_puerto:container_puerto**

```
docker run -p 5000:8080 spring-boot-websocket-chat-demo
```

Si la aplicación es necesario que corra desacoplada, se puede ocupar el flag -d

```
docker run -d -p 5000:8080 spring-boot-websocket-chat-demo
```

Para listar los contenedores.

```
docker container ls
```

Para iniciar sesión en Docker.

```
docker login
```

Para subir la imagen a Docker Hub (Plataforma mantenida por Docker que contiene imágenes de contenedores), primero es necesario crear un tag (repositorio remoto). la sintaxis de este comando es.

```
docker tag image username/repository:tag
```

Luego para subir la imagen.

```
docker push username/repository:tag
```

Finalmente, para traer cambios de dicha imagen, y correrlas de manera local.

```
docker run -p host_port:container_port username/repository:tag
```

Por otro lado, para eliminar imágenes creadas.

```
docker rmi Image1
```

En caso de que en nuestro núcleo Docker, hayan imágenes inútiles, mediante dangling podemos eliminarlas.

```
docker rmi $(docker images -f dangling=true -q)
```

Para detener un contenedor Docker.

```
docker stop CONTAINER_ID
```