Contenedor de su aplicación de arranque Spring con Docker

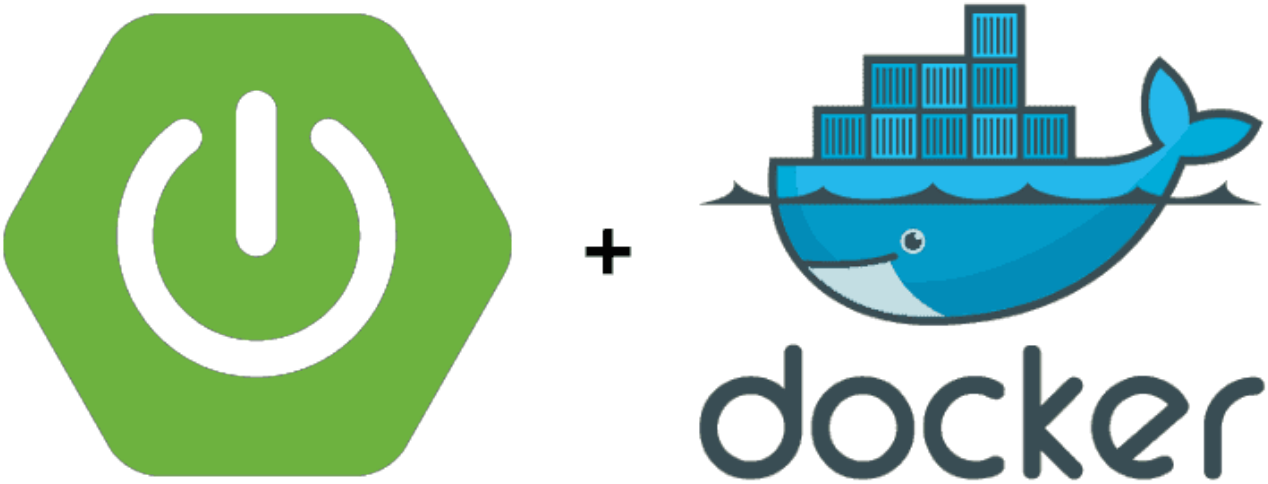
5 de febrero de 2019 | 9 min de lectura | Docker , Maven , Spring , Spring Boot , Web

Hace aproximadamente dos años, escribí una publicación en el blog sobre contener su aplicación de arranque Spring con Docker. Sin embargo, algunas cosas han cambiado, y dentro de este tutorial le daré una versión más actualizada para contener sus aplicaciones de arranque Spring.

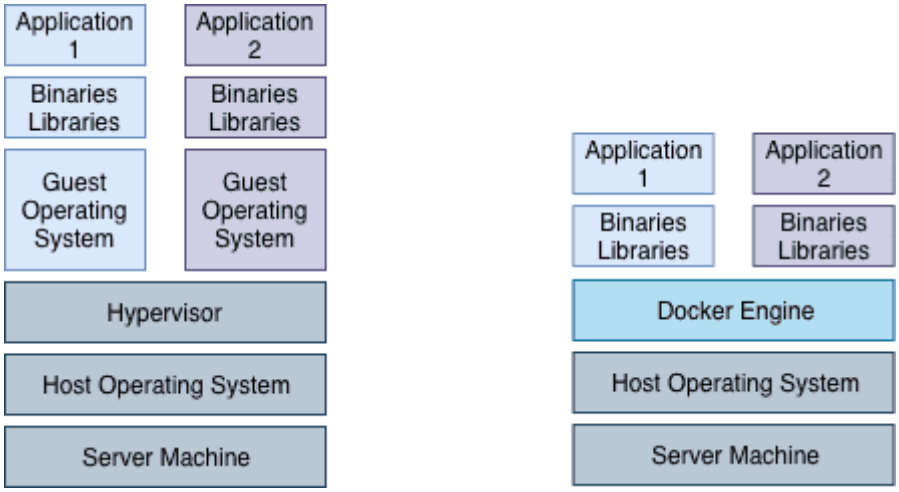
¿Qué es Docker?

En caso de que aún no haya oído hablar de Docker, Docker es una plataforma abierta para construir, enviar y ejecutar aplicaciones envolviéndolas en contenedores.

Gracias a Docker, los problemas como "funciona en mi máquina" pueden pertenecer al pasado, ya que todos ejecutarán su aplicación en un entorno similar, arrancado por Docker.



Quizás se pregunte, ¿en qué se diferencia todo esto del uso de máquinas virtuales? Bueno, una diferencia es que los contenedores Docker no se ejecutan en un sistema operativo separado. En su lugar, utilizan el motor Docker para comunicarse con el sistema operativo host.
Esto es más conveniente en términos de recursos, ya que no requiere recursos adicionales para configurar un sistema operativo invitado completo.



El Dockerfile

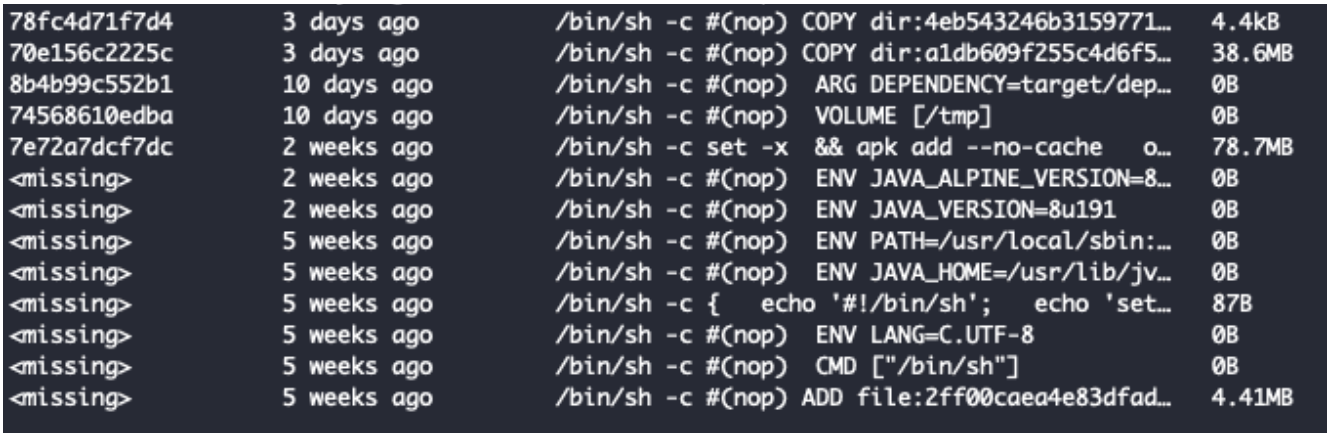
Otra diferencia es que no tiene que compartir una máquina virtual completa con otros para que puedan ejecutar su aplicación. En lugar de eso, crea un archivo que contiene un conjunto de operaciones llamado **Dockerfile** .

Este archivo contiene una lista de operaciones que se utilizan para ensamblar una imagen de Docker. Estas operaciones describen los comandos necesarios para poder ejecutar su aplicación desde cero. Cada una de estas operaciones da como resultado una "capa" separada para su imagen de Docker.

La ventaja de este sistema en capas es que Docker solo procesa capas que cambian cuando intenta actualizar una imagen, o extraer una imagen para su uso, que es más rápido que tener que construir todo una y otra vez.

Con el `docker history` comando, puede averiguar qué capas contiene una imagen de Docker.

```
dimitri at dimitris-mbp in ~
$ docker history g00glen00b/movie-quote-service
IMAGE               CREATED             CREATED BY                                      SIZE
d19a7e33ef74        3 days ago          /bin/sh -c #(nop)  ENTRYPOINT ["java" "-XX:+…   0B
d9fc3124fa36        3 days ago          /bin/sh -c #(nop) COPY dir:03d186d5338f775ae…   11.2kB
```



```
78fc4d71f7d4        3 days ago          /bin/sh -c #(nop) COPY dir:4eb543246b3159771…   4.4kB
70e156c2225c        3 days ago          /bin/sh -c #(nop) COPY dir:a1db609f255c4d6f5…   38.6MB
8b4b99c552b1        10 days ago         /bin/sh -c #(nop)  ARG DEPENDENCY=target/dep…   0B
74568610edba        10 days ago         /bin/sh -c #(nop)  VOLUME [/tmp]                0B
7e72a7dcf7dc        2 weeks ago         /bin/sh -c set -x  && apk add --no-cache   o…   78.7MB
<missing>           2 weeks ago         /bin/sh -c #(nop)  ENV JAVA_ALPINE_VERSION=8…   0B
<missing>           2 weeks ago         /bin/sh -c #(nop)  ENV JAVA_VERSION=8u191       0B
<missing>           5 weeks ago         /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:…   0B
<missing>           5 weeks ago         /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jv…   0B
<missing>           5 weeks ago         /bin/sh -c {   echo '#!/bin/sh';   echo 'set…   87B
<missing>           5 weeks ago         /bin/sh -c #(nop)  ENV LANG=C.UTF-8             0B
<missing>           5 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>           5 weeks ago         /bin/sh -c #(nop) ADD file:2ff00caea4e83dfad…   4.41MB
```

Desempaca tu JAR

Dado que cada capa se procesará individualmente, sería más interesante si separara su JAR en varias capas, una para sus clases y otra para sus dependencias.

Si echa un vistazo a la captura de pantalla anterior, la capa que contiene sus clases es de solo **11,2kb** , que es mucho menos que la capa que contiene las dependencias, que es **38,6Mb** . Eso significa que reconstruir la imagen de Docker cuando se produce un cambio de código será mucho más rápido si lo comparas con no tener capas separadas para tu JAR y tener que reconstruir esa capa en su lugar.

Para poder escribir un Dockerfile que utilice estas capas separadas, primero tenemos que descomprimir el archivo JAR generado. Para este propósito, **usaremos el complemento de dependencia de maven** para hacer esto.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>unpack</id>
            <phase>package</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
```

```
                    <artifactItem>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>${project.artifactId}</artifactId>
                        <version>${project.version}</version>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Limitaciones de memoria

Un problema que puede encontrar al ejecutar su aplicación de arranque Spring dentro de un contenedor Docker es que se queda sin memoria y el contenedor está siendo eliminado.

Para resolver este problema específico, debemos asegurarnos de que la JVM esté al tanto de las restricciones de memoria que existen en el contenedor. Una opción es usar **cgroup** , que es una característica dentro del kernel de Linux para saber qué límites hay.

Para decirle a la JVM que queremos usar esto, debemos asegurarnos de que estamos usando al menos Java **8u131** , luego podemos usar los siguientes parámetros:

```
java \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+UseCGroupMemoryLimitForHeap \
  -XX:MaxRAMFraction=1 \
  -XshowSettings:vm \
  -version
```

Si está ejecutando esto dentro de un contenedor Docker, verá que el tamaño máximo de almacenamiento dinámico que menciona es casi idéntico a la cantidad de memoria que se le da al contenedor. No está bien documentado, pero parece

que también tiene en cuenta la memoria que no es del montón, por lo que permitir una fracción máxima de RAM del 100% no debería suponer ningún problema.

Definiendo un Dockerfile

Ahora que hemos desempaquetado nuestro JAR y somos conscientes de cómo limitar adecuadamente la JVM, podemos crear nuestro propio Dockerfile:

```
FROM openjdk:8-jre-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
EXPOSE 8080
ENTRYPOINT ["java","-XX:+UnlockExperimentalVMOptions","-XX:+UseCGroupMemor
```

Lo que sucede aquí es que usamos las carpetas generadas por el **complemento de dependencia de maven** y las copiamos individualmente a nuestra imagen de Docker en ubicaciones específicas. Además, pasamos las opciones de Java que se mencionaron anteriormente a la JVM, además de agregar nuestro classpath y decirle a la JVM qué clase ejecutar.

Usando Maven para construir tu imagen Docker

El último paso que podemos tomar es integrar la construcción de nuestra imagen Docker dentro de las herramientas que ya usamos para construir aplicaciones relacionadas con Java, que en mi caso, es Maven.

Un complemento que puede ayudarlo con esto es el **dockerfile-maven-plugin** . Esto no solo le permitirá construir sus imágenes de Docker, sino también llevarlo a un registro de Docker para que otros puedan usarlo.

Por ejemplo, si desea crear una imagen y usar la ID del artefacto como nombre y etiquetarla con la versión dentro de su archivo **pom.xml** , puede usar algo como

esto:

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.4.9</version>
    <executions>
        <execution>
            <id>build</id>
            <phase>package</phase>
            <goals>
                <goal>build</goal>
            </goals>
            <configuration>
                <repository>g00glen00b/${project.artifactId}</repository>
                <tag>${project.version}</tag>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Esto, al empaquetar la aplicación, generará una imagen Docker para su proyecto utilizando el Dockerfile dentro de su proyecto. Por defecto, se verá en el directorio base de su proyecto (el mismo directorio donde se encuentra el **pom.xml**). Sin embargo, esto puede cambiarse configurando la `contextDirectory` propiedad.

Si realiza un cambio en sus clases ahora y reconstruye el proyecto usando Maven, puede ver que solo cambió esa capa específica dentro del historial de Docker:

```
dimitri at dimitris-mbp in ~/Downloads/movie-quote-app/movie-quote-service on master [+!]
$ docker history g00glen00b/movie-quote-service:0.0.1
IMAGE               CREATED             CREATED BY                                      SIZE
a595fc158029        3 seconds ago       /bin/sh -c #(nop)  ENTRYPOINT ["java" "-XX:+…   0B
1637cc94ba62        3 seconds ago       /bin/sh -c #(nop)  EXPOSE 8080                  0B
1246973280db        3 seconds ago       /bin/sh -c #(nop) COPY dir:366730ca21403db83…   11.4kB
a35c8fa6b93c        3 minutes ago       /bin/sh -c #(nop) COPY dir:3e50fd50593abe565…   4.35kB
d21328d72117        3 minutes ago       /bin/sh -c #(nop) COPY dir:672ee1446d1250f63…   40.7MB
```

En esta captura de pantalla, puede ver que he compilado la aplicación dos veces. Primero hice una compilación inicial, hace **3 minutos** y todas las capas tuvieron

que construirse.

La segunda vez (hace **3 segundos**), hice un pequeño cambio en mi código (pero no en las dependencias), y puede ver que solo **recreó** algunas capas, la primera es la capa de imagen **1246973280db** , que contiene mi código
Además, desencadenó una reconstrucción en todas las capas dependientes, que son las capas que contienen my `EXPOSE` y `ENTRYPOINT` command.

Trabajando con Docker Compose

Una característica interesante adicional del ecosistema Docker es Docker Compose . Docker Compose permite a un usuario ejecutar ciertos contenedores, simplemente al proporcionar un archivo de configuración YAML llamado `docker-compose.yml` . Por ejemplo, para ejecutar una aplicación, puede usar:

```
version: '3.7'
services:
  movie-quote-service:
    image: g00glen00b/movie-quote-service:0.0.1
    ports:
      - 8080:8080
```

Con esta configuración simple, puede ejecutar un contenedor específico y exponer un puerto específico a la red host, de modo que pueda acceder a él a través de http: // localhost: 8080 . Para ejecutar este ejemplo, puede usar el `docker-compose up` comando.

También le permite vincular fácilmente contenedores, por ejemplo:

```
version: '3.7'
services:
  movie-quote-service:
    image: g00glen00b/movie-quote-service:0.0.1
    ports:
      - 8080:8080
    depends_on:
      - movie-quote-database
```

```
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://movie-quote-database/quotes?use
      - SPRING_DATASOURCE_USERNAME=dbuser
      - SPRING_DATASOURCE_PASSWORD=dbpass
  movie-quote-database:
    image: mysql:8.0.14
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_USER=dbuser
      - MYSQL_PASSWORD=dbpass
      - MYSQL_DATABASE=quotes
      - MYSQL_ONETIME_PASSWORD=true
```

En este ejemplo, también configuré un contenedor MySQL como una base de datos. Como solo servirá como fuente de datos para el servicio, no necesito exponer ningún puerto a la máquina host.

Sin `depends_on` embargo, tengo que configurar la propiedad para asegurarme de que el contenedor de la base de datos esté disponible desde el contenedor de la aplicación. Después de hacer eso, puede acceder al otro contenedor utilizando el nombre del otro contenedor como su nombre de host. En este ejemplo eso sería `jdbc:mysql://movie-quote-database`.

Si no desea ingresar estas contraseñas de texto sin formato dentro de su `docker-compose.yml` archivo, puede crear un `.env` archivo y asegurarse de no comprometerlo en alguna parte. Por ejemplo, podría agregar:

```
DATABASE_USER=dbuser
DATABASE_PASSWORD=dbpass
DATABASE_ROOT_PASSWORD=password
DATABASE_NAME=quotes
```

Después de eso, puede reemplazar estos valores dentro de su `docker-compose.yml` por `${DATABASE_USER}` ...

```
version: '3.7'
services:
  movie-quote-service:
```

```
    image: g00glen00b/movie-quote-service:0.0.1
    ports:
      - 8080:8080
    depends_on:
      - movie-quote-database
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://movie-quote-database/${DATABASE
      - SPRING_DATASOURCE_USERNAME=${DATABASE_USER}
      - SPRING_DATASOURCE_PASSWORD=${DATABASE_PASSWORD}
  movie-quote-database:
    image: mysql:8.0.14
    environment:
      - MYSQL_ROOT_PASSWORD=${DATABASE_ROOT_PASSWORD}

      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_ONETIME_PASSWORD=true
```

Persistiendo en el host

Un problema con la configuración actual es que una vez que destruya el contenedor de su base de datos y lo vuelva a crear, se perderán todos los datos. Una solución a este problema es asignar un volumen específico dentro de su `docker-compose.yml` archivo, por ejemplo:

```
version: '3.7'
services:
  movie-quote-database:
    image: mysql:8.0.14
    environment:
      - MYSQL_ROOT_PASSWORD=${DATABASE_ROOT_PASSWORD}
      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_ONETIME_PASSWORD=true
    volumes:
      ./data/:/var/lib/mysql
```

En este caso, todos los datos relevantes almacenados en la base de datos se almacenarán en la carpeta **./data** , en relación con su proyecto.

Intercomunicación bidireccional de contenedores

Hasta ahora, ya hemos visto cómo su contenedor de aplicaciones puede comunicarse con su contenedor de base de datos `depends_on` . Sin embargo, en algunos casos, ambos contenedores tienen que comunicarse entre sí.

Como las dependencias circulares no están permitidas, tenemos que encontrar una solución diferente, y esa solución es agregar su propia red. Por ejemplo:

```
version: '3.7'
services:
  movie-quote-service:
    image: g00glen00b/movie-quote-service:0.0.1
    ports:
      - 8080:8080
    depends_on:
      - movie-quote-database
    networks:
      movie-quote-network:
        aliases:
          - movie-quote-service
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://movie-quote-database/${DATABASE
      - SPRING_DATASOURCE_USERNAME=${DATABASE_USER}
      - SPRING_DATASOURCE_PASSWORD=${DATABASE_PASSWORD}
  movie-quote-database:
    image: mysql:8.0.14
    networks:

      movie-quote-network:
        aliases:
          - movie-quote-database
    environment:
      - MYSQL_ROOT_PASSWORD=${DATABASE_ROOT_PASSWORD}
      - MYSQL_USER=${DATABASE_USER}
```

```
      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_ONETIME_PASSWORD=true
networks:
  movie-quote-network:
```

Como puede ver, hemos definido una red llamada **movie-quote-network** en la parte inferior, con un cuerpo vacío ya que solo voy a confiar en la configuración predeterminada. Además de eso, también agregué una `networks` sección a cada configuración de contenedor y proporcioné un alias para ese contenedor específico.

Ambos contenedores podrán comunicarse entre sí utilizando el alias dado como nombre de host. En este caso, hemos establecido el alias igual al nombre del contenedor, por lo que nada más cambia dentro de nuestra configuración.

Volver a los tutoriales • Contáctame en Twitter • Comenta en Twitter

Dimitri "g00glen00b" Mestdagh es consultor en Cronos y líder tecnológico en Aquafin. Por lo general, puede encontrarlo probando nuevas bibliotecas y tecnologías. Ama tanto Java como JavaScript.