Wynn Teo Follow

Aug 24, 2021 · 3 min read · · Listen

Dockerizing a Spring Boot Application with Maven

In the previous post, we created Spring Boot REST API that runs locally.

Before we create the Dockerfile, we need to modify the pom.xml file to include the build plugin that will be executed during the build lifecycle. We need this to generate the executable jar file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.4.5</version>
                <relativePath />
                <!-- lookup parent from repository -->
        </parent>
        <groupId>springboot.api</groupId>
        <artifactId>spring-boot-api-tutorial</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>spring-boot-api-tutorial</name>
        <description>Demo project for Spring Boot REST API</description>
        <!-- define the packaging type -->
        <packaging>jar</packaging>
        <properties>
                <java.version>11</java.version>
                <maven.test.skip>true</maven.test.skip>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
```

```
47                 <plugins>
48                         <plugin>
49                                 <groupId>org.springframework.boot</groupId>
50                                 <artifactId>spring-boot-maven-plugin</artifactId>
51                                 <executions>
52                                         <execution>
53                                                 <configuration>
54                                                         <mainClass>springboot.api.tutorial.TutorialApplication</mainClass>
55                                                 </configuration>
56                                         </execution>
57                                 </executions>
58                         </plugin>
59                 </plugins>
60         </build>
61 </project>
```

pom.xml hosted with ❤ by GitHub view raw

Now, create a **Dockerfile** in the project root folder.

```
1  # AS <NAME> to name this stage as maven
2  FROM maven:3.6.3 AS maven
3  LABEL MAINTAINER="sgwebfreelancer@gmail.com"
4
5  WORKDIR /usr/src/app
6  COPY . /usr/src/app
7  # Compile and package the application to an executable JAR
8  RUN mvn package
9
10 # For Java 11,
11 FROM adoptopenjdk/openjdk11:alpine-jre
12
13 ARG JAR_FILE=spring-boot-api-tutorial.jar
```

```
19
20  ENTRYPOINT ["java","-jar","spring-boot-api-tutorial.jar"]
```

Dockerfile hosted with ❤ by GitHub view raw

Notice that we are using two **FROM** in the Dockerfile, we called it **multi-stage builds**. Multi-stage builds can help to **optimize** the docker image. We copy the built jar file from stage one which is maven and store only the jar file in the current working directory. Then, we discard the local Maven repositories and class files generated in the target directory.

Next, create the **docker-compose.yml** file in the project root folder.

```
1   version: '3'
2
3   services:
4     mysqldb:
5       container_name: mysqldb
6       image: mysql:latest
7       environment:
8         - MYSQL_ROOT_PASSWORD=P@ssw0rd
9         - MYSQL_DATABASE=spbapidb
10        - MYSQL_USER=user
11        - MYSQL_PASSWORD=password
12      restart: always
13      ports:
14        - 3306:3306
15      volumes:
16        - mysqldb:/var/lib/mysql
17    api:
18      container_name: api
19      image: spring-boot-api-tutorial-img
20      build:
21        context: ./
22        dockerfile: Dockerfile
23      depends_on:
24        - mysqldb
25      ports:
26        - 8080:8080
27      restart: always
28
29  volumes:
30    mysqldb:
```

docker-compose.yml hosted with ❤ by GitHub view raw

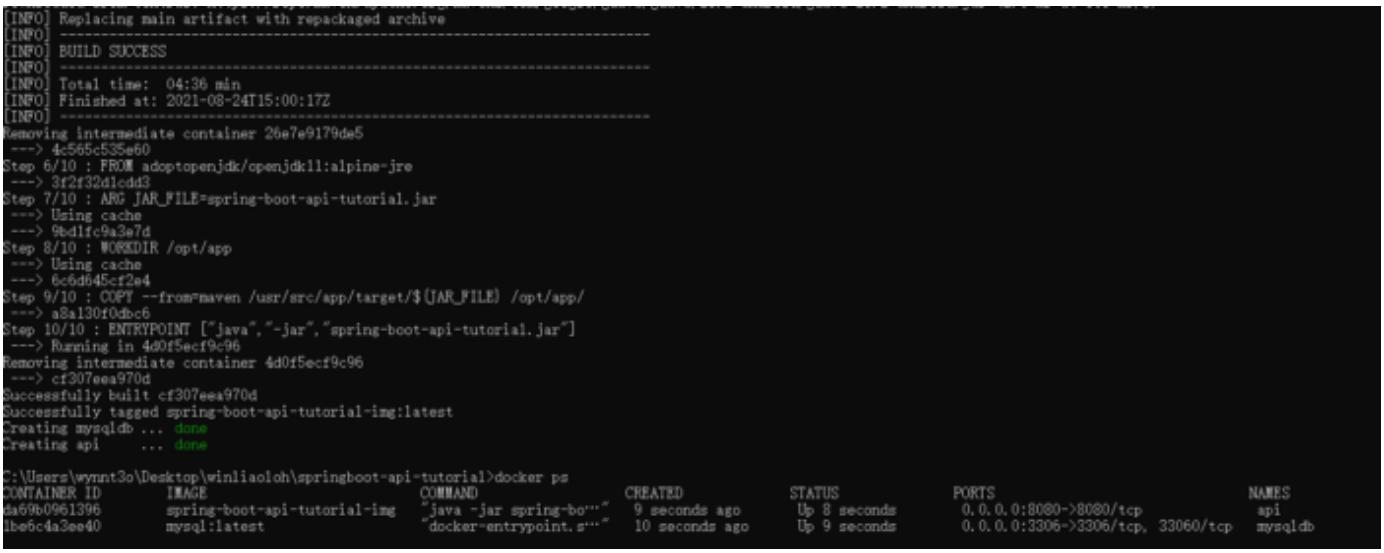
Lastly, since we are creating MySQL inside the docker container, we have to update the data source URL.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://mysqldb:3306/<MYSQL_DATABASE>?allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username=<MYSQL_USER>
spring.datasource.password=<MYSQL_PASSWORD>
server.port = 8080
```

All is ready! Run the command below to build the docker.

```
docker-compose up --build

docker ps
```



Try to access the URL *http://192.168.99.100:8080/api/products*. You should see an empty array.

```
[]
```

Startup Scripts

The official MySQL docker image will run `.sql` scripts found in the `/docker-entrypoint-initdb.d/` folder. Files will be executed in alphabetical order. You can easily populate your `mysql` services by mounting a SQL dump into that directory.

```
version : '3'

services:
  mysql:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_ROOT_PASSWORD: P@ssw0rd
    ports:
      - "3306:3306"
    volumes:
      - "./db/script/setup.sql:/docker-entrypoint-initdb.d/setup.sql"
```

Then, run the command below to connect to MySQL and insert a product record.

```
mysql -u USERNAME -pPASSWORD DATABASENAME

INSERT INTO products (id, title, description, price, created_at, updated_at) VALUES(1,'Apple', 'This is an
apple.', 3.5, now(), now());
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21
Server version: 8.0.26 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SHOW TABLES;
+-------------------+
| Tables_in_tutorial |
+-------------------+
| products          |
+-------------------+
1 row in set (0.00 sec)

mysql> INSERT INTO products (id, title, description, price, created_at, updated_at) VALUES(2,'Orange', 'This is an orange.', 2.5, now(), now());
Query OK, 1 row affected (0.01 sec)

mysql>
```

Now try to access the URL *http://192.168.99.100:8080/api/products* again. You should be able to see the product records.

```
[{"id":1,"title":"Apple","description":"This is an apple.","price":3.5,"created_at":"2021-08-24T15:10:44.000+00:00","updated_at":"2021-08-
24T15:10:44.000+00:00"},{"id":2,"title":"Orange","description":"This is an orange.","price":2.5,"created_at":"2021-08-
24T15:13:18.000+00:00","updated_at":"2021-08-24T15:13:18.000+00:00"}]
```

Now, end the MySQL connection. Let's explore the volume we have created for MySQL. You should see all the data files were stored here.

```
cd var/lib/mysql
```

```
root@1be6c4a3ee40:/# cd var/lib/mysql
root@1be6c4a3ee40:/var/lib/mysql# ls
' #ib_16384_0.dblwr'    auto.cnf    binlog.index   client-cert.pem   ib_logfile0   ibtmp1      performance_schema   server-cert.pem   tutorial
' #ib_16384_1.dblwr'    binlog.000001   ca-key.pem     client-key.pem    ib_logfile1   mysql       private_key.pem      server-key.pem    undo_001
' #innodb_temp'         binlog.000002   ca.pem         ib_buffer_pool    ibdata1       mysql.ibd   public_key.pem       sys               undo_002
root@1be6c4a3ee40:/var/lib/mysql#
```

Remember we said the volume is created to persist the data so that when we re-run the docker, it won't wipe out the MySQL data. Let's stop the ***mysqldb*** and ***api*** containers. Then restart the containers again.

```
docker stop mysqldb
docker stop api

docker-compose up -d
```

Thanks for reading.

Build a Spring Boot REST API with Java, Maven, and MySQL
In this tutorial, we learn how to build a Spring Boot Rest API with Maven and MySQL. Spring Boot helps you accelerate...
medium.com

Deploy Spring Boot App into AWS Elastic Beanstalk using AWS CLI
So we have created Spring Boot REST API that running locally. We are now going to deploy the Spring Boot application...
aws.plainenglish.io