Spring Boot Application build and run

Written by Hong Le Nguyen, Last updated 05.2025
Related Code : https://github.com/hong1234/springOnDockerV1

You can build and run Spring Boot applications in several ways

Running the application from the command line using the Maven or Gradle

```
mvn spring-boot:run
gradle bootRun
```

Using Maven or Gradle to build a Spring Boot application as an executable JAR file that can be run at the command line or be deployed in the cloud => 1)

Packaging (using Docker or Maven/Gradle Plugin) a Spring Boot application as a Docker container image that can be deployed to any platform that supports Docker containers, including Kubernetes environments => 2)

Using Maven or Gradle to build a Spring Boot application as a WAR file that can be deployed to a traditional Java application server such as Tomcat => 3)

Note:
When we use Spring Initializr (https://start.spring.io), one of the entries included in our pom.xml file was spring-boot-maven-plugin. This plugin hooks into Maven's package phase and does a few extra steps.

```
// pom.xml
<plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

1) Build, run App as JAR on JKD

building an JAR file

```
mvn clean package
```

This Maven command has two parts:
clean: Deletes that target folder and any other generated outputs. This is always good to include before building an uber JAR to ensure all generated outputs are up-to-date.
package: Invokes Maven's package phase, which will cause the verify, compile, and test phases to be invoked in the proper order.

Spring Boot simply lets our code run as if the third-party JAR files are there as always. No shading is required. *The app is ready to run wherever you have a JDK installed.*

Once the application is packaged as a JAR, run the application as follows:

```
java -jar app.jar
```

You can have multiple profile configuration files application-*.properties, such as

```
applicationqa.properties       // default
application-prod.properties
```

and you can activate the desired profiles using the spring.profiles.active system property.

```
java -jar app.jar -Dspring.profiles.active=prod
```

Spring Boot allows you *to override the configuration parameters* in various ways. You can override the properties using system properties as follows:

```
java -jar app.jar -Dserver.port=8585
```

2) Running a Spring Boot Application on Docker

2.1) Running a Spring Boot Application in a Docker Container

step1--
// src\main\resources\application.properties

step2 --
Create Dockerfile in the root of the project, as follows:

```
// Dockerfile
FROM openjdk:17.0.2-jdk
ADD target/springboot-demo.jar app.jar
RUN bash -c 'touch /app.jar'
ENTRYPOINT ["java","-jar","/app.jar", "-Djava.security.egd=file:/dev/./urandom"]
```

This code uses openjdk:17.0.2-jdk as the base image. It copies target/springboot-demo.jar into the target image with the name app.jar. Finally, it invokes the java -jar app.jar command using ENTRYPOINT.

step3 --
Before building the Docker image, you first need to build the application.

mvn clean package

Now you can run the docker build command as follows:

docker build -t hongle/springboot-demo .

Here, you are tagging the image with the name hongle/springboot-demo. Since you haven't activated any profiles, the default profile will be active and the application will use the H2 in-memory database.

Step4 --
Now launch the container from the hongle/springboot-demo image, as follows.

```
docker run -d \
--name springboot-demo \
-p 80:8080 \
hongle/springboot-demo
```

You are running the container by giving it the name springboot-demo and exposing the container's port 8080 on the Docker host machine at port 80.

2.2) Running Multiple Containers

Now you'll see how to launch a Postgres database in one Docker container and then launch the application in another container using the Postgres database from the first container.

step1 --
Now you create the Docker profile configuration file **application-docker.properties**, as follows:

```
// src\main\resources\application-docker.properties
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://${POSTGRES_SERVER}/${ POSTGRES_DB}
spring.datasource.username=${POSTGRES_USER}
spring.datasource.password=${POSTGRES_PASSWORD}
// spring.datasource.initialize=true
spring.jpa.hibernate.ddl-auto=update
```

step2 --
You can modify the Dockerfile to run the application by activating the docker profile.

```
// Dockerfile
FROM openjdk:17.0.2-jdk
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar", "-Dspring.profiles.active=docker"]
```

Note that the code specifies -Dspring.profiles.active=docker to activate the docker profile so that the application will use the Postgres database running in another Docker container instead of using the H2 in-memory database.

step3 --
Now build the project using the mvn clean package command

```
mvn clean package
```

Now build your application Docker image.

```
docker build -t hongle/springboot-demo .
```

step4 --
You can launch a Postgres database container by using the postgres image, as follows:

```
docker run --name demo-postgres \
-e POSTGRES_SERVER= postgres-docker:5432 \
-e POSTGRES_DB=demodb \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=postgres \
-d postgres
```

This code launches a postgres container in detached mode by using the -d flag and gives it the name demo-postgres. It also specifies the database name, username, and password as environment variables by using -e flags.

Before launching your container, you need to delete the existing container. You can remove the existing container as follows:

```
docker rm springboot-demo
```

You need to link the application's Docker container with the demo-postgres Docker container in order to be able to use the Postgres database from the application.

Launch the application container using the following command:

```
docker run -d \
--name springboot-demo \
--link demo-postgres:postgres \
-p 80:8080 \
hongle/springboot-demo
```

Note that you linked the demo-postgres container using the --link flag and gave it an alias called postgres. Now the application is running in one container and is talking to the Postgres database running in another container.

2.2-B) Running Multiple Containers Using docker-compose

step1, step2, step3 as above

If your application depends on multiple services and you need to start all of them in Docker containers, it is tedious to start them individually.

You can use the docker-compose tool to orchestrate the multiple containers required to run your application.

Step4-B --

You need to create a docker-compose.yml file and configure the services that you want to run.
Create docker-compose.yml in the root directory of your application, as shown in Listing

```
// docker-compose.yml
version: '3.8'
services:

  demo-postgres:
    image: postgres:latest
    container_name: postgres-docker
    volumes:
      - product-data:/var/lib/postgresql/data
    ports:
      - 5432:5432
    environment:
      - POSTGRES_DB=demodb
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    networks:
      - ecom-network

  springboot-demo:
    image: hongle/springboot-demo
    container_name: springboot-docker
    depends_on:
      - "demo-postgres"
    environment:
      - POSTGRES_SERVER=postgres-docker:5432
    ports:
      - 80:8080
    networks:
      - ecom-network

volumes:
  product-data:

networks:
  ecom-network:
```

Now you can simply run the docker-compose up command from the directory where you have the docker-compose.yml file to start the application and the Postgres containers.

```
docker-compose up
```

docker down

```
docker-compose down
```

removing images

```
docker rmi -f hongle/springboot-demo
docker rmi -f postgres
```