**SPRING_APP_DEPLOY**

Last Updated 05.2025

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

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

You can have multiple profile configuration files, such as

```
applicationqa.properties
application-prod.properties
```

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

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

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

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

**/// Running a Spring Boot Application on Docker ///**

**1)**
Running a Spring Boot Application in a Docker Container

step1--
Before building the Docker image, you first need to build the application.

```
springboot-heroku-demo> mvn clean package
```

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

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

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

Now you can run the docker build command as follows:

```
docker build -t sivaprasadreddy/springboot-heroku-demo .
```

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

step3--
Now launch the container from the sivaprasadreddy/springboot-heroku-demo image, as follows.

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

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

**2)**
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:

```
// application-docker.properties
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://
${POSTGRES_PORT_5432_TCP_ADDR}:${POSTGRES_PORT_5432_TCP_PORT}/demodb
spring.datasource.username=${POSTGRES_ENV_POSTGRES_USER}
spring.datasource.password=${POSTGRES_ENV_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
ADD target/springboot-heroku-demo.jar app.jar
RUN bash -c 'touch /app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=docker",
"-jar","/app.jar"]
```

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
```

step4--
Now build your application Docker image.

```
docker build -t sivaprasadreddy/springboot-heroku-demo .
```

step5--

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

```
docker run --name demo-postgres \
-e POSTGRES_DB=demodb \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=secret123 \
-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:

```
sudo docker rm springboot-heroku-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-heroku-demo \
--link demo-postgres:postgres \
-p 80:8080 \
sivaprasadreddy/springboot-heroku-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.

**2B)**
**Running Multiple Containers Using docker-compose**

step1,...,step4 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.

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 16-14.

```
// docker-compose.yml
version: '3.8'
services:
  demo-postgres:
      image: postgres:latest
      environment:
        - POSTGRES_DB=demodb
        - POSTGRES_USER=postgres
        - POSTGRES_PASSWORD=secret123

  springboot-heroku-demo:
      image: sivaprasadreddy/springboot-heroku-demo
      links:
        - demo-postgres:postgres
      ports:
        - 80:8080
...
```

step5B--
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
```