**Dockerizing a SpringBoot application**

In this workshop are going to dockerize a Spring Application with an embedded H2 database.

Next, we are going to connect a dockerized Spring Application with an external dockerized MySQL database.

**Prerequisites**

- Have Docker installed

This tutorial makes use of:

- IntelliJ IDE

- Java 1.8 SDK

- PostMan

**Let’s get started**

The application we are going to use in this tutorial is the whiskeyshop application. This application can be retrieved from GitHub <https://github.com/AMIS-Services/sig-springboot-1>

Our application has the basic CRUD-operations. Create, read, update & delete.

Spring Boot makes it especially easy to use an in-memory database, because it can create the configuration automatically for H2, HSQLDB (HyperSQL), and Apache Derby.

All we need to do to use a database of one of the three types in Spring Boot is add its dependency to the pom.xml. In the example application we use H2 as an embedded in-memory database.

To easily deploy our application on any platform. We are going to dockerize our application.

**Creating a Dockerfile**

A Dockerfile specifies how a new Docker Image gets built. The following Dockerfile is defining a Docker image for out SpringBoot Application:

# Start with a base image containing Java runtime

FROM openjdk:8-jdk-alpine

# Add Maintainer Info

LABEL maintainer="amis"

# Application runs on port 8080

EXPOSE 8080

# The application's jar file

ARG JAR\_FILE=target/whiskeyshop-0.0.1-SNAPSHOT.jar

# Add the application's jar to the container

ADD ${JAR\_FILE} whiskeyshop.jar

# Run the jar file

ENTRYPOINT ["java","-jar","/whiskeyshop.jar"]

Let’s walk through the syntax:

-**FROM**: A Docker Image can use another image as a base and add functionality on top of that. We are going to run our application by executing a **jar-file** and thus need a java runtime environment. We can simply get our desired environment by starting with the base image **openjdk:8-jdk-alpine** and adding our jar-file to the environment. The base image we used is a very lightweight OpenJDK 8 runtime image that uses Alpine Linux.

-**LABEL**: The **LABEL** instruction is used to add metadata to the image. In this case we added some info about the maintainer of the image.

-**EXPOSE**: The **EXPOSE** instruction serves as a documented suggestion to the user running the container which ports should be published. When a container is run, the exposed ports are not actually published. Running a container with the **-p** flag lets you manually map the host and container ports. Running a container with the **-P** flag will publish all exposed ports to the host interface. (We’ll also see the run commands in a later section)

-**ARG**: The **ARG** instruction lets you define variables, which you can refer to later in your Dockerfile. Used for better organizing & adding extra readability to your Dockerfile.

-**ADD**: The **ADD** instruction is used to copy new files and directories to the docker image.

-**ENTRYPOINT**: The **ENTRYPOINT** instruction specifies which command to run on startup of the container.

For more info about Dockerfiles, please refer to <https://docs.docker.com/engine/reference/builder/>

Place the Dockerfile in your SpringBoot application, on the same level as pom.xml.

**Building a Docker Image**

Now that we have defined the Dockerfile, let’s build a Docker Image for the application.

First, we need to make sure that the package we are going to add to our Docker Image exists.

Build the application, with either of the following command:

mvn clean package

./mvnw clean package

./mvnw.cmd clean package

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Side note:***

Adding the maven-wrapper to your projects lets the application be more standalone. As your users do not need to separately install maven.

Generating an application with the Spring Initializr (<https://start.spring.io/>) will automatically create a Maven Wrapper into your project.

Or add one manually, with the following command:

mvn -N io.takari:maven:wrapper

For more info about the Maven Wrapper, refer to <https://github.com/takari/maven-wrapper>. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

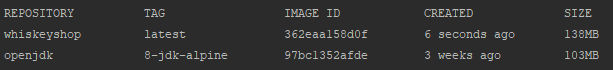
Let’s build a Docker Image for the application.

We can do this by executing the following command at the root level of your application:

docker build -t whiskeyshop .

To list all the docker images on your system, execute the following:

docker image ls



The **-t** flag let us name and optionally tag the image in the form **name:tag**, to more easily refer to it later. By default, the tag will be **latest**, and can be left out. You can explicitly give a tag as a version for your created image.

Whenever you make an adjustment to the application and want to update the image, you need to build the package again and build the image again.

**Running a Docker Image**

Let’s get to know some commands first so we can play around with containers and test our application.

Once you have a Docker Image, you can create a container (a running instance of an image):

docker run -p 9000:8080 whiskeyshop

The **-p** flag publishes the mapped ports. The first is the host port, the second is the container port.

To run a docker image in detached mode, execute the following:

docker run -d -p 9000:8080 whiskeyshop

To list all containers (running instances of images), execute the following:

docker container ls

Or to list all containers, use the equivalent:

docker ps

To stop all docker containers, execute the following:

docker stop $(docker ps -q)

Docker nowadays uses structured commands for more clarity. Use this also to your advantage when wanting more info about commands, with the **--help** flag.

docker --help

docker run --help

docker container --help

docker container ls --help

Alright, so now your application should be running a Docker container. Let’s test if the application also responds by firing some requests.

Your application in the container should be running at localhost:9000

Try a Get-request a the /whiskeys path.

This should return an empty list.

Look at the controller in the application and try some CRUD-operations on the listed Whiskey.

Tip, Whiskey-products are in the form of:

{  
 "id": 1,  
 "name": "Jack Daniels"  
}

You should be able to see added items to your list,

After you add some whiskey-products with POST-requests, you should be able to see your update list at the /whiskeys path with a GET request.

You should also be able to see the changes you have made with the other CRUD-operations.

Congratulations you just dockerized a SpringBoot application!

***Side note***:

When you make changes to your application’s code and want to update your application in the Docker container, you should first update the Docker image and then redeploy a new running instance of this image. Don’t forget to stop your old container which was running.

**Part2**

**Running a database in a separate Docker container**

For our external database we are going to use a MySQL database. For many … there already exist docker images on Dockerhub. We can easily make use of these already made images by simply pulling and running them.

By executing the following the command:

(if the image was not found locally, then it will be pulled from dockerhub)

docker run -e MYSQL\_ROOT\_PASSWORD=pass -e MYSQL\_DATABASE=test -p 3306:3306 -d mysql

For more info about the image, and options to run refer to <https://hub.docker.com/_/mysql/>.

With **docker ps**, you should be able to see the container running. But how do we let our application in one container connect to the MySQL database in the other container?

**host.docker.internal** can be used as an alternative for localhost in your datasource-url.

Outside of a docker container you would connect to a container via **localhost:PublishedPort**.

Inside of your container you can use **host.docker.internal** to refer to the internal IP address used by the host.

In the example application, there is an **application-mysql.properties** with the following properties:

spring.datasource.url=jdbc:mysql://host.docker.internal:3306/test?allowPublicKeyRetrieval=true&useSSL=false  
spring.datasource.username=root  
spring.datasource.password=pass  
  
spring.jpa.hibernate.ddl-auto=create

The options **spring.datasource.username** and **spring.datasource.password** are the username and password used to access the data source.

With the option **spring.jpa.hibernate.ddl-auto** you can let Spring take care of creating a schema (table) for your database. For embedded databases this will be **create-drop** by default, and thus will also wipe the content of the database when starting and closing your application. For other external databases the default setting is **none**, as you do not want Spring to change the database structure unprompted. So, be careful adjusting this property when you are not in a development environment.

Since no table exists yet when we run a new MySQL container, we let Spring create one for us. Spring will base the schema of the Whiskey Entity we have defined in our application.

Once a table has been created in the MySQL database, you can set the property back to none or delete the line. This will also make your data persistent, until you remove the MySQL container. Remember, that you have to redeploy your docker container though if you want to update your application in the docker container.

The **mysql-properties** are set in a different file, how do we let Spring know to actually use them? In Spring you can have different profiles. For example, you could a have a development profile and a production profile. In the development profile you want to quickly test with an H2 embedded database, while in production you want to connect to an external MySQL database.

You can set up different application-{profile}.properties files and set the environment variable **spring.profiles.active** equal to the activated profile. The activated profile will overwrite the properties of the default **application.properties** file.

To run the container with the mysql-profile, execute the following command:

docker run -dp 9000:8080 -e spring.profiles.active="mysql" whiskeyshop

Different property values for the **spring.datasource.url** and **spring.datasource.password** can also be passed as environment variables.

Now it’s time to fire some CRUD-request at your application and see if your application works.

Also try to let your data to be more persistent. Make your data stay even if you stop your application container. (Hint adjust the spring.jpa.hibernate.ddl-autoproperty)

You now have 2 docker containers interact with each other. Congratulations!

**Cleanup**

If you are done and want to clean up some docker images and docker containers. Here are some commands.

Your docker containers might be stopped, but not yet removed.

Take a look at them with:

docker container ls -a

To remove all non-running containers, execute the following:

(Running containers need to be stopped first, before removing them)

docker rm $(docker ps -aq)

To remove all the docker images, execute the following command:

(This will remove all images without at least one container associated to them)

docker image prune -a

**Appendix - files**

**application-mysql.properties**

spring.datasource.url=jdbc:mysql://host.docker.internal:3306/test?allowPublicKeyRetrieval=true&useSSL=false  
spring.datasource.username=root  
spring.datasource.password=pass  
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect  
  
spring.jpa.hibernate.ddl-auto=create