**Lab 4  
REST web service in a Docker container**

This lab will show you how you can run your Spring Boot REST service in a Docker container.

On top of that, we will verify how secure the container is, using an open source tool named Claire.

The starting point for this lab is to have the provided VirtualBox machine up-and-running:

* You are logged in under user/password: developer/welcome01
* You have updated the labs running the git pull command in the lab workspace directory /home/developer/projects/SIGSpringBoot101

Note that in this lab, we will NOT use Eclipse STS.

# Building the Docker container

We will use the service that we developed in lab 3. You can find the completed code in:

/home/developer/projects/SIGSpringBoot101/lab 4/dronebuzzers

The following steps will be taken:

* Step 1: change the Maven pom.xml  
  The Maven pom.xml has to be changed so it can create a Docker container
* Step 2: add the Dockerfile  
  The Dockerfile contains the Docker container definition
* Step 3: build the container  
  Use Maven to create a Docker container

**Step 1: change the Maven pom.xml**

The Maven file pom.xml will have to undergo two changes:

1. Add a property ‘docker.image.prefix’, that specifies the Docker image prefix name
2. Add the Spotify plugin for building the Docker container

@1:add the property line in the <properties> tag:

<docker.image.prefix>docker</docker.image.prefix>

@2: add the plugin in the <plugins> tag:

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.3.4</version>

<configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>

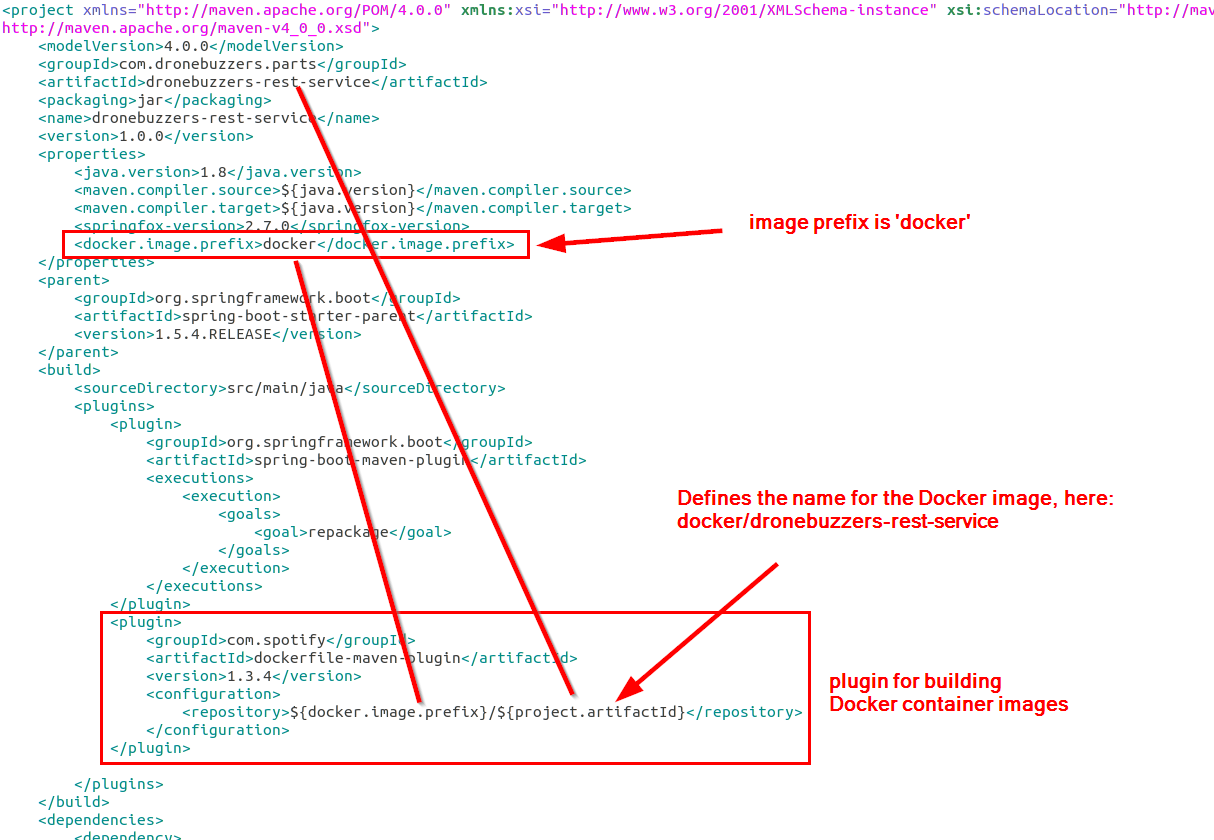
</configuration>

</plugin>

The resulting pom.xml file can also be found in:

/home/developer/projects/SIGSpringBoot101/lab 4/input/pom.xml

Examining the pom.xml file:



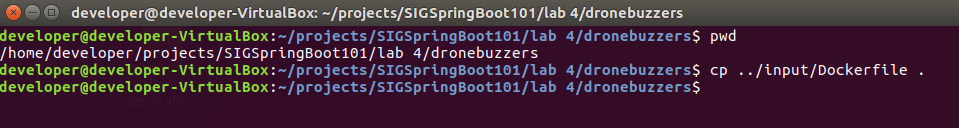
**Step 2: add the Dockerfile**

In order to build a Docker container, we need a file named Dockerfile that specifies the container.

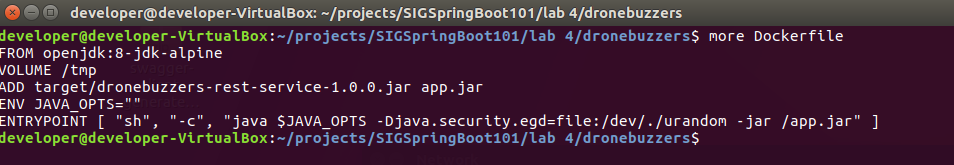
Copy the Dockerfile from the *input* to the project directory *lab 4/dronebuzzers*:

In a terminal window, navigate to the directory lab 4/dronebuzzers. Then perform a copy action

cp ../input/Dockerfile .



Have a look at the Dockerfile:



The keywords in the Dockerfile have the following meaning:

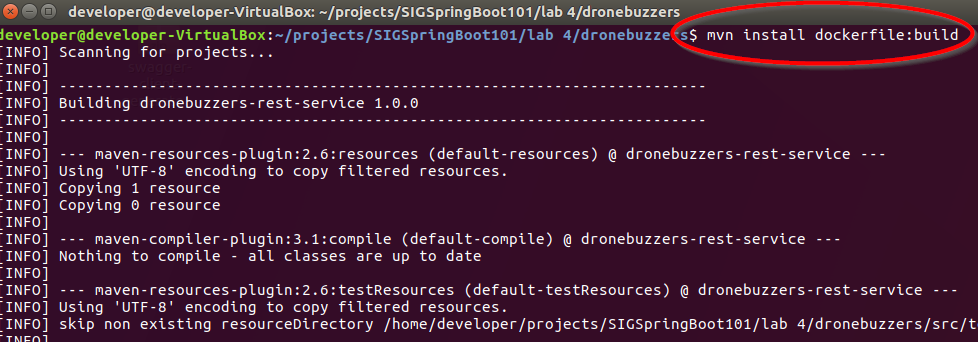
|  |  |
| --- | --- |
| Keyword | Meaning |
| FROM | sets the base image for the container. This means that our container will be built ‘on top of‘ the FROM container |
| VOLUME | external mounted volume. This makes the indicated volume write its data on the host machine |
| ADD | adds files/directories from the source to the container image |
| ENV | environment variable |
| ENTRYPOINT | command that the container will run when started |

So for our container:

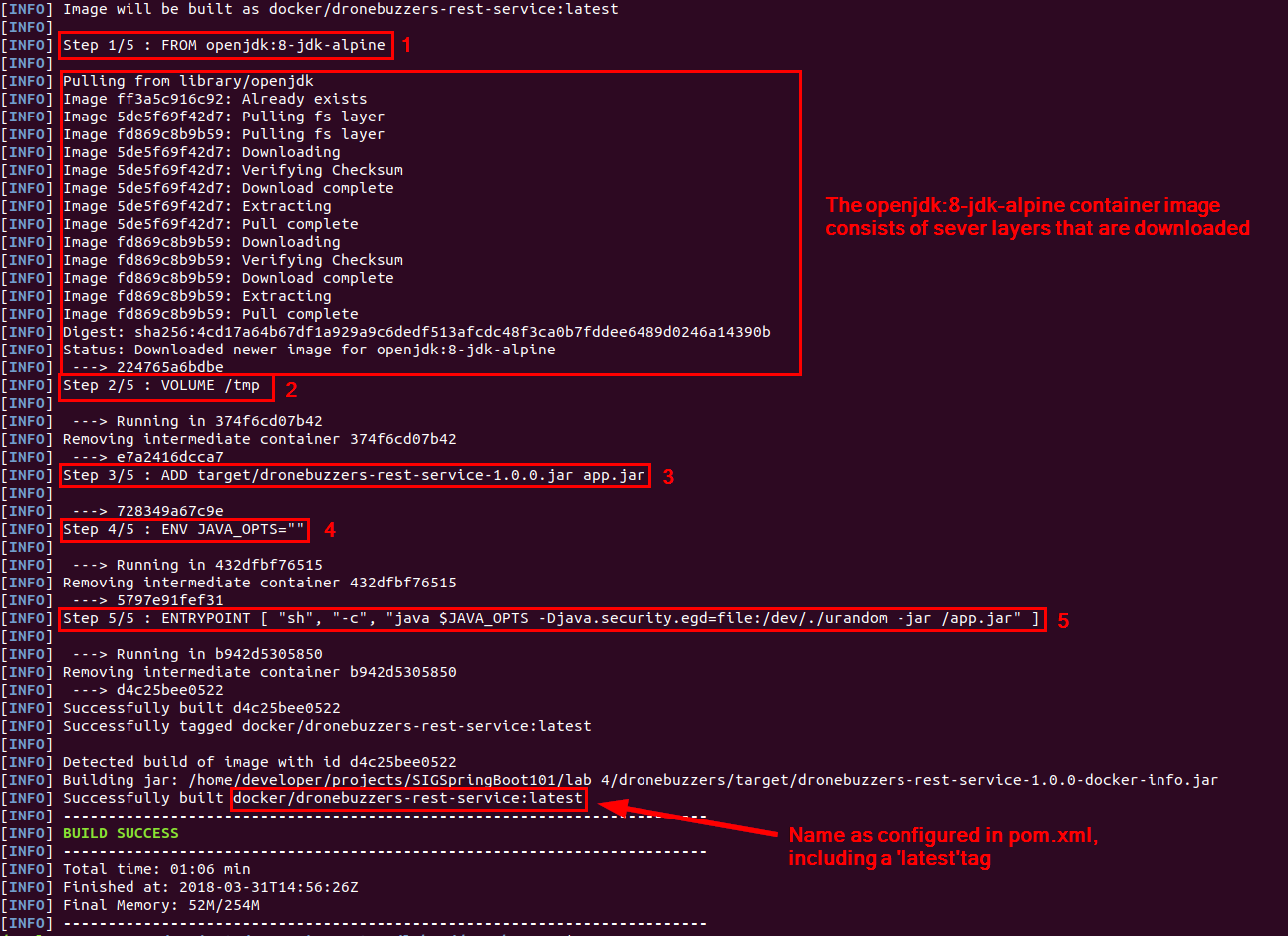
|  |  |  |
| --- | --- | --- |
| Keyword | Value | Meaning |
| FROM | openjdk:8-jdk-alpine | base image for the container is openjdk:8-jdk-alpine |
| VOLUME | /tmp | The container’s /tmp volume will write its data to the host. Unless explicitly defined otherwise during start-up of the container, the volume will end up in /var/lib/docker/volumes |
| ADD | target/dronebuzzers-rest-service-1.0.0.jar app.jar | adds the Spring Boot application to the container image under the name app.jar |
| ENV | JAVA\_OPTS="" | Sets the environment variable JAVA\_OPTS to “” in the container |
| ENTRYPOINT | [ "sh", "-c", "java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ] | Upon start, the container will run a shell that executes our application (app.jar) |

**Step 3: build the container**

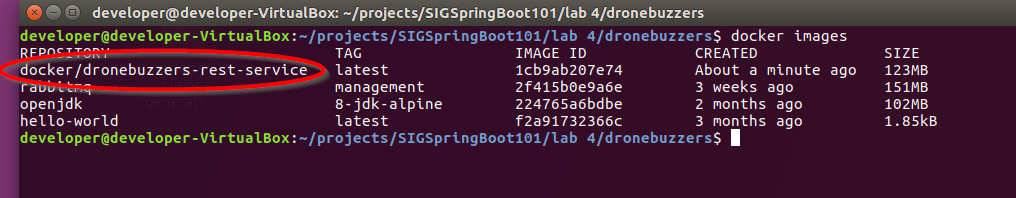
The Docker container image is built with the command mvn install dockerfile:build



Near the end of the execution of the command, the Docker file execution is clearly visible:

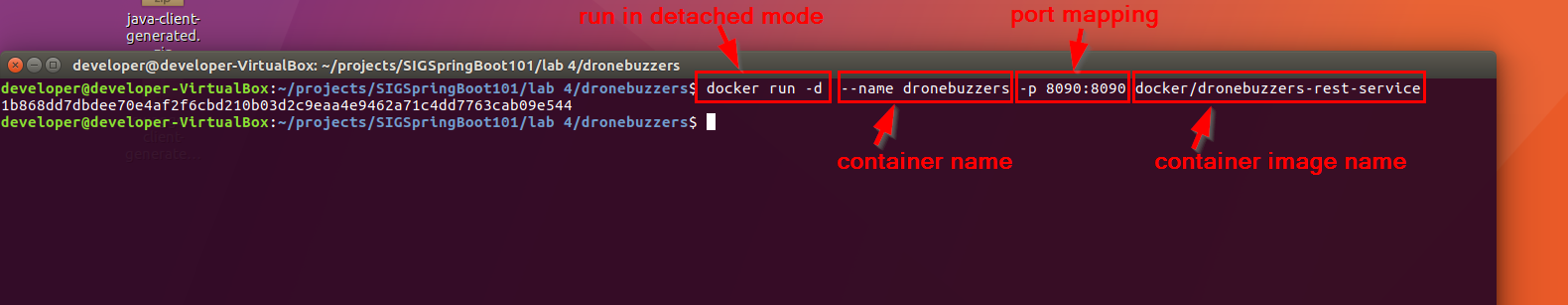


Check that the Docker container image has been created with the command docker images:



# Running the Docker container

Now, the container that we’ve built can be started with the docker run command:



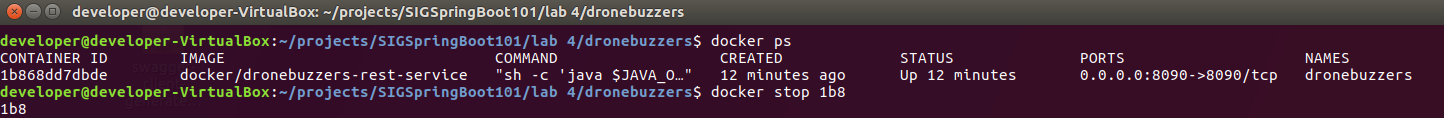
The command to be used:

docker run -d --name dronebuzzers -p 8090:8090 docker/dronebuzzers-rest-service

* -d: the container runs in de-tached mode, i.e. in the background
* -p 8090:8090: specifies port exposure, i.e. how a local host port (first 8090)is mapped to the internal container port (second 8090)

Note: if you still have a Spring Boot application running from Eclipse, listening to port 8090, then you will not be able to run the container; in that case, either stop the application in Eclipse to change the first 8090 in the command above in some other port number.

Should you later on want to stop the container: look up the Container ID with the *docker ps* command. Then stop the container with the *docker stop* command. Notice that you only have to enter the first couple of characters of the Container ID.



There is a good Docker cheat sheet: <https://github.com/wsargent/docker-cheat-sheet>

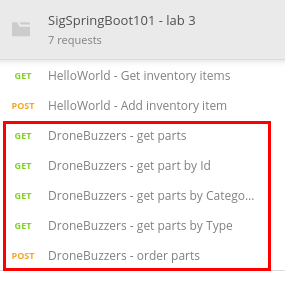
# Testing the Docker container

Once the container is up and running, we can test it. We will do that with the same Postman tests from lab 3.

For testing, start Postman  and import the Collection of Postman tests for lab 3 from location

/home/developer/projects/SIGSpringBoot101/lab 3/postman

Test the interface with the last 5 requests in the Postman collection:



The responses in Postman will not look any different now they come from the SpringBoot application running inside a Docker container compared with before when the application was executed from within Eclipse.

At this moment, the container is running inside the same VM as Eclipse. However, we can move that container virtually anywhere (where we can run a Docker container) and access the REST API there.