



## **DISTRIBUTED SYSTEMS**

# Deployment using Docker

Tudor Cioara  
Dan Mitrea

Cristina Pop  
Alex Rancea

Marcel Antal  
Liana Todorean

2023-2024

## Contents

1. What is Docker? Why choosing Docker over VMs? .....	3
1.1. How to install Docker .....	4
1.2. Docker container lifecycle .....	4
Docker REGISTRY .....	5
Create Docker Image .....	5
Create Docker Volume .....	5
Configure Network .....	6
Create Container .....	6
Check Running container and logs .....	7
Stop/Remove Container and Image .....	7
Enter terminal of a container .....	7
1.3. Docker Commands Summary .....	7
2. DOCKER deploy example .....	9
2.1. Deployment of Spring application .....	9
Test your solution .....	11
2.2. Deployment of React application .....	12
Test your solution .....	12
3. Application Deployment Diagram .....	14
References .....	16

## 1. What is Docker? Why choosing Docker over VMs?

The Operating System divides the computer memory in several sections, where the **Kernel space** and the **User space** are most important, as shown in Figure 1. The **Kernel Space** is the portion of memory where privileged operating system kernel processes are executed, while the User Space contains unprivileged processes. The separation is performed using a set of privileges. Programs or processes are run in user mode and are sandboxed, meaning that they are isolated from other processes from the memory point of view and cannot have complete access to the computer's memory, disk storage, network hardware, and other resources.

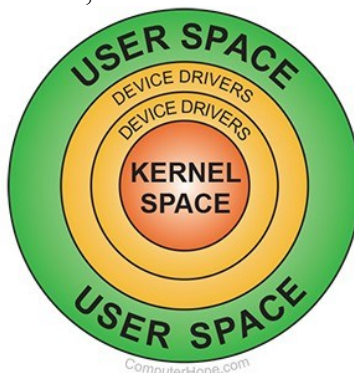


Figure 1. Computer Memory privileges separation (Source [8]).

Starting with 1970, IBM and later other companies started developing special software called **Hypervisor**, used to create and run Virtual Machines (VMs) [9]. A VM is an emulation of a computer system, based on a computer architecture and providing the functionality of a physical computer [15]. Several VMs with different hardware requirements and guest OSes can be run on a host computer. Multiple instances of a variety of operating systems may share the virtualized hardware resources: for example, Linux, Windows, and macOS instances can all run on a single physical x86 machine. The hypervisor runs in **Kernel mode**, while the **guest OS** runs in **user mode**, thus theoretically being **sandboxed** from the host OS.

As opposed to virtualization, **Docker** is a container-based technology where containers are running as processes in the user space of the operating system. Docker originally used Linux Containers (LXC), but later switched to runC (formerly known as libcontainer), which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. At the low level, a container is just a set of processes that are isolated from the rest of the system, running from a distinct image that provides all files necessary to support the processes. It is built for running applications. In Docker, the containers running share the host OS kernel.

Table 1. VM and Docker comparison [10,11,12]

Features	VM	Docker
Host OS	Any	Linux – based (if installed on Windows it installs a VM with Linux)
Guest OS	Any	Linux – based (it uses the kernel of the host operating system)
Sandboxing	Full isolation	Can access host through shared filesystem (such as docker-volume)
HW Resource requirements	High (similar to the physical machine emulated)	Low (many containers can run on the same physical machine)

1.1. How to install Docker

Recommended to use Linux. Docker for Windows seems to create a VM with Linux on it, on which it runs Docker, inside which it then runs containers. So much indirection might lead to problems in the future (such as managing volumes). Furthermore, **docker** commands need **privileged** access, using **sudo** command. Follow the tutorial here and install Docker CE [7]: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>

1.2. Docker container lifecycle

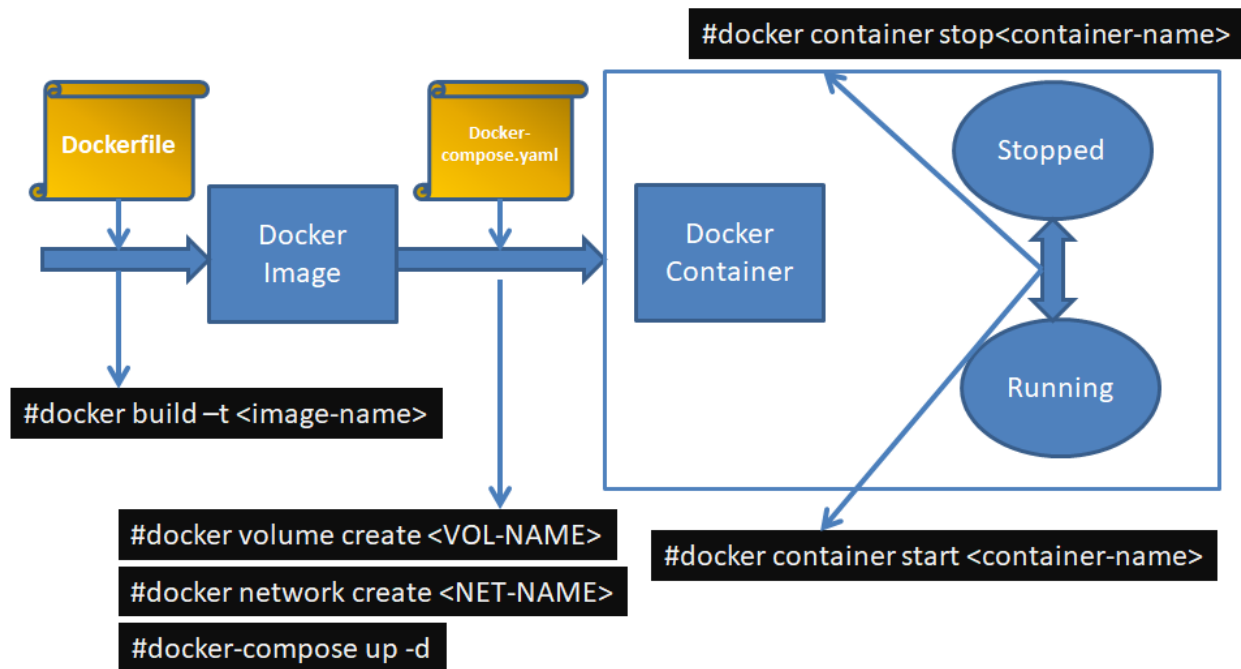


Figure 2. Docker lifecycle and basic commands

The first step would be to create an empty folder for each docker container you want to create. This folder should contain at least 2 files, with exactly these names (since only these are recognized by docker). The files will be detailed in the following sections.

- Dockerfile – the description of the image
- Docker-compose.yaml – the description of the container.

### Docker REGISTRY

The Docker registry [13] is a server application that stores and distributes Docker images. Almost all custom images that will be build are based on an existing image that already exists in the Docker Registry [14]. Images can also build from scratch, without inheriting any parent image.

### Create Docker Image

The Docker image is described by a *Dockerfile*. An image can be created either by inheriting a parent image, or by creating a base image from scratch [14]:

- **Inherit a parent image:** the new image will customize an existing image (parent image) from Docker Registry, by referencing it using the FROM directive at the beginning of the Dockerfile. All the other instructions in the docker file modify the parent image.
- **Create a base image:** a base image is created by using the FROM scratch directive at the beginning of the Dockerfile.

Thus, a docker image can either be used directly from an online repository (e.g. the Docker REGISTRY) or it can be customized starting from a parent image by describing it in a Dockerfile and issuing the following command in the terminal:

```
#docker build -t <image-name>
```

An example for a Dockerfile for Nodejs server (downloaded from Docker REGISTRY) customized for deploying an application is the following:

```
FROM node:8
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
CMD npm start
EXPOSE 3000
```

By issuing the docker build command, an image with the name <image-name> will be created. At this stage, the image is stored locally and can be viewed using the command:

```
#docker image ls
```

### Create Docker Volume

To create a docker volume you just have to run:

```
#docker volume create <vol name>.
```

The information from the volume will be stored at `/var/lib/docker/volumes/<vol_name>/_data`. You can copy any files of interest directly here and they will appear in the container in the specified folder (see Figure 3).

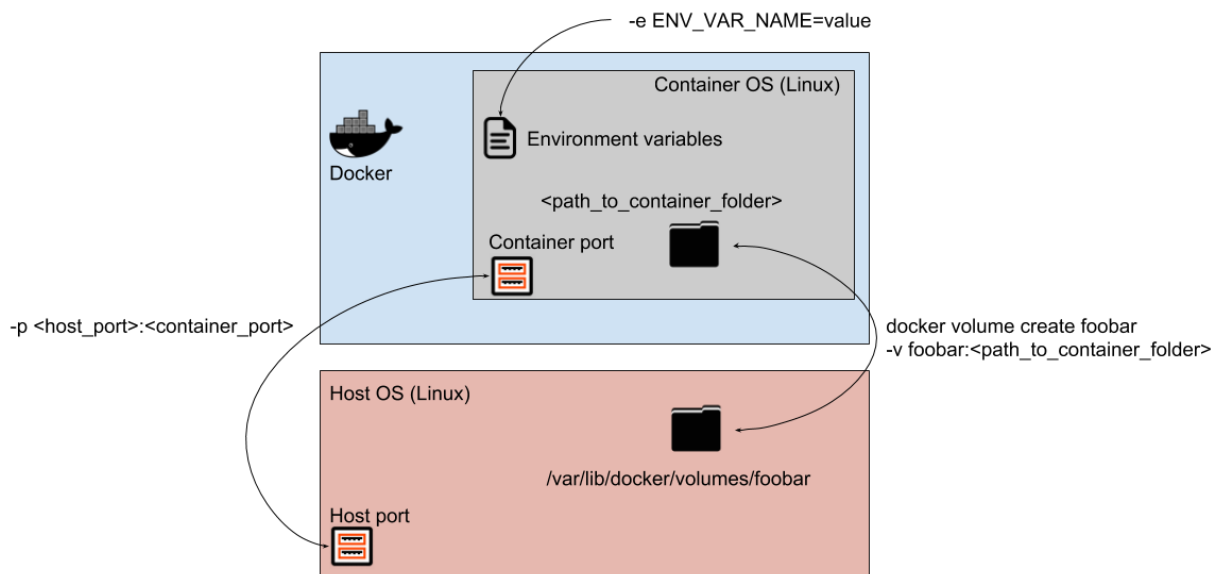


Figure 3. A docker container and communication with the host (port mapping and shared folders using volumes)

To see all available volumes, you can run `docker volume ls`. To remove a volume you can run `docker volume rm <vol_name>`.

### Configure Network

A virtual network can be created between the containers. Each network has a name and a IP address. To create a network, the following command is used:

```
#docker network create <NET-NAME>
```

### Create Container

There are 3 basic things that we want to synchronize between a docker container and the host:

- ports: map a container's port to a host's port (to be able to access the application from outside)
- volumes: map a container's folder to a host's Docker volume (to be able to persist information and do backups)
- environment vars: set the container's environment variables that can be used by the application for various configurations

You can use docker-compose files to write the configurations you want for a docker container. A docker-compose file MUST be named `docker-compose.yaml`, and for the syntax you can check the existing ones or the internet.

The docker-compose files for the most used images are already created, and you can find them in Docker-compose Scripts folder at this location (mysql, cassandra, tomcat, gitlab).

In order to run/create a docker container you have to move this docker-compose.yaml file to the desired computer, cd to its location and run:

```
#docker-compose up -d
```

Make sure the appropriate docker volumes are created before running the above command (e.g. tomcat volume for tomcat image, check the used volumes in each docker-compose file in particular).

Containers created and started with this are started and stopped with the classic Docker commands (start, stop, restart).

#### Check Running container and logs

The containers that are running can be checked with

```
#docker ps
```

The logs of a container can be accessed using

```
#docker logs -f <container-name>
```

#### Stop/Remove Container and Image

Containers and corresponding images can be stopped and removed using the following commands (in this order):

```
#docker container stop <container-name>  
#docker container rm <container-name>  
#docker image rm <image-name>
```

#### Enter terminal of a container

One can enter the terminal of a container using the following command:

```
#docker exec -it <container-name> /bin/bash
```

Furthermore, using the instruction `docker exec -it <container-name>`, other commands to the container can be appended.

### 1.3. Docker Commands Summary

Interrogate running containers:

```
> docker ps
```

Interrogate existing volumes:

```
> docker volume ls
```

Stop and Remove running containers:

```
> docker stop {name}
```

```
> docker rm {name}
```

Remove existing volume:

```
> docker volume rm {volume-name}
```

Use docker-compose.yml files, if the container needs a volume create it first with:

```
> docker volume create {volume-name}
```

Then start the docker container using:

```
> docker-compose up -d
```

Volumes location on host:

```
/var/lib/docker/volumes/
```

Access container bash:

```
docker exec -it [container-id] /bin/bash
```



## 2. DOCKER deploy example

In this section we will exemplify the deployment on Docker of the Spring demo application and React application from tutorials for Assignment 1. The code can be downloaded from:

- Spring Application [1]: git clone [https://gitlab.com/ds\\_20201/spring-demo.git](https://gitlab.com/ds_20201/spring-demo.git)
- React Application [3]: git clone [https://gitlab.com/ds\\_20201/react-demo](https://gitlab.com/ds_20201/react-demo)

### 2.1. Deployment of Spring application

In order to be able to build a custom docker image containing your application’s executable code a Dockerfile must be configured in the root directory of the project.

A Dockerfile and docker-compose.yml files are already available in the Spring demo if you switch the branch to **docker\_production**. Otherwise, create a new branch and create yourselves the files specified in the laboratory work.

```
#git fetch -a
#git branch -a
#git checkout docker_production
```

The Dockerfile used for building the image for our application is presented in Figure 5.

Starting with line 1, an intermediate maven image called builder is configured. This is used in order to build the executable of the application from the source code. Thus, firstly the source code is copied in the temporary image (src, pom.xml and checkstyle.xml). Normally, a Spring Boot application can be started using this .jar file. **However, due to the boot resources limitation, we have considered a more optimal approach, by using layered Jars.** More details about the Layered Jars and the reasons for using them can be found at [2].

1. In order to specify that a layered jar is required, firstly you need to modify lines 93-97 from pom.xml file, and add the following configuration, specifying that layers are enabled.

```
88 <build>
89 <plugins>
90 <plugin>
91 <groupId>org.springframework.boot</groupId>
92 <artifactId>spring-boot-maven-plugin</artifactId>
93 <configuration>
94 <layers>
95 <enabled>true</enabled>
96 </layers>
97 </configuration>
98 </plugin>
```

Figure 4. Section from pom.xml of spring-demo

2. The **mvn package** is run (Figure 5 line 7) in order to obtain the application’s layered .jar file from the source code.
3. The layers of the created .jar file are listed using command at line 8
4. The layers of the created .jar file are extracted using command at line 9
5. Lines 14-19 from the Dockerfile presented in Figure 5 contain the DB credentials

```

FROM maven:3.8.3-openjdk-17 AS builder

COPY ./src/ /root/src
COPY ./pom.xml /root/
COPY ./checkstyle.xml /root/
WORKDIR /root
RUN mvn package
RUN java -Djarmode=layertools -jar /root/target/ds-2020-0.0.1-SNAPSHOT.jar list
RUN java -Djarmode=layertools -jar /root/target/ds-2020-0.0.1-SNAPSHOT.jar extract
RUN ls -l /root

FROM openjdk:17-jdk-slim-buster

ENV TZ=UTC
ENV DB_IP=localhost
ENV DB_PORT=3306
ENV DB_USER=root
ENV DB_PASSWORD=local-server1
ENV DB_DBNAME=city-db

COPY --from=builder /root/dependencies/ ./
COPY --from=builder /root/snapshot-dependencies/ ./

RUN sleep 10
COPY --from=builder /root/spring-boot-loader/ ./
COPY --from=builder /root/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher", "-
XX:+UseContainerSupport -XX:+UnlockExperimentalVMOptions -
XX:+UseCGroupMemoryLimitForHeap -XX:MaxRAMFraction=1 -Xms512m -
Xmx512m -XX:+UseG1GC -XX:+UseSerialGC -Xss512k -XX:MaxRAM=72m"]

```

Figure 5 Dockerfile for Spring Boot Application

Starting with line 12, a JDK 17 image is used and the layers obtained in the previous temporary image are copied in this image (lines 22, 23, 26, 27). Furthermore, the details for the DB connections can be specified in the Dockerfile as environmental variables. In this way, the source code will not contain the connection details but will be able to read them from the Environmental Variables set in the Dockerfile. This is possible by having the right configuration in your spring-boot *application.properties* file. Each variable from the *application.properties* has a format of `${ENV_VAR_NAME: default_value}` specifying that, if the ENV\_VAR\_NAME is found in the local environmental variables, then that value is considered, otherwise the default value is considered. On one hand, in the development mode (when the project is setup in your IDE) there are no environmental variables set, so the default values are considered. On the other hand, when

the docker image is launched, the environmental variables are set (Figure 6 lines 15-19) so the specified values are considered, and the default ones are ignored.

```

1 #####
2 ### DATABASE CONNECTIVITY CONFIGURATIONS ###
3 #####
4 database.ip = ${DB_IP:localhost}
5 database.port = ${DB_PORT:5432}
6 database.user = ${DB_USER:root}
7 database.password = ${DB_PASSWORD:root}
8 database.name = ${DB_DBNAME:city-db}
9

```

Figure 6. Application Properties section from spring-demo

Line 28 from the Dockerfile from Figure 5, specifies the command with which the newly created image should be launched. As noticed, there are several options set in order to optimize the resources consumption during boot time.

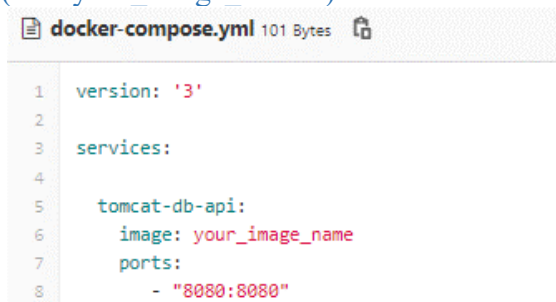
**Remember to set the `spring.jpa.hibernate.ddl-auto` property to create/validate/update according to your database structure and contents.**

Test your solution

Before continuing your configuration on the Gitlab repository, make sure that your *Dockerfile* written by you is correct and that the obtained image can run successfully.

For this follow the instructions:

1. Create a *docker-compose.yml* file in the root of your project, containing the following lines. The name of the image must correspond to the name given as argument to the build command from step 2 (i.e. “your\_image\_name”)



```

1 version: '3'
2
3 services:
4
5   tomcat-db-api:
6     image: your_image_name
7     ports:
8       - "8080:8080"

```

Figure 7. Docker-compose file of spring-demo image

2. Build your image using:
  - `docker build -t your_image_name .`
3. Start your image:
  - `docker-compose up -d`
4. Access your deployed application at `http://localhost:8080`. Furthermore, other endpoints of the application should be accessible, such as `http://localhost:8080/person`, returning the list of persons stored in the DB.

If everything is successful, you can push your newly created files on your repository (create a new branch like the example given *docker\_production* branch) and proceed with the Gitlab configuration.

## 2.2. Deployment of React application

For the Frontend application, you can find a React application configured to be built and deployed using docker [3]. The docker based CI/CD is configured on the **docker\_production** branch.

### Observations:

- Check the Dockerfile exposed on the **docker\_production** branch:



```
1 FROM node:8 as builder
2 WORKDIR /app
3 COPY package*.json /app/
4 RUN npm install
5 COPY ./ /app/
6 RUN npm run build
7
8
9 FROM nginx:1.17-alpine
10 RUN apk --no-cache add curl
11 RUN curl -L https://github.com/a8m/envsubst/releases/download/v1.1.0/envsubst-`uname -s`-`uname -m` -o envsubst && \
12     chmod +x envsubst && \
13     mv envsubst /usr/local/bin
14 COPY --from=builder /app/nginx.conf /etc/nginx/nginx.template
15 CMD ["/bin/sh", "-c", "envsubst < /etc/nginx/nginx.template > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'" ]
16 COPY --from=builder /app/build/ /usr/share/nginx/html
```

Figure 8. Dockerfile for docker production branch

Same approach is used as for the Maven project. In the first stage the application is built in an intermediate node image, while the built results are copied in the final nginx image. The Envsubst plugin is installed in order to make possible the parametrization of the nginx scripts with Environmental Variables. More details about this at [6].

- A nginx.conf file must be added in the root directory in order to specify the configuration for the nginx server



```
1 server {
2     listen      ${PORT:80};
3     server_name _;
4
5     root /usr/share/nginx/html;
6     index index.html;
7
8     location / {
9         try_files $uri /index.html;
10    }
11 }
```

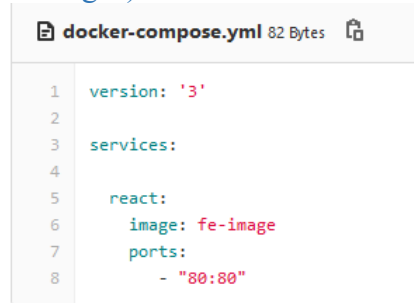
Figure 9. nginx.conf configuration file

Test your solution

Before continuing your configuration on the Gitlab repository, make sure that the Dockerfile written by you is correct and that the obtained image can run successfully.

For this follow the instructions:

1. Create a *docker-compose.yml* file in the root of your project, containing the following lines. The name of the image must correspond to the name given as argument to the build command from step 2 (i.e. “fe-image”)



```
1 version: '3'
2
3 services:
4
5   react:
6     image: fe-image
7     ports:
8       - "80:80"
```

Figure 10. Docker-compose configuration file for React app

2. Build your image using:
  - `docker build -t fe-image .`
3. Start your image:
  - `docker-compose up -d`
4. Access your deployed application at <http://localhost>

If everything is successful, you can push your newly created files on your repository (create a new branch like the example given *docker\_production* branch) and proceed with the Gitlab configuration.

### 3. Application Deployment Diagram

The application composed of the backend, frontend and database will be deployed in docker as showed by the following architecture:

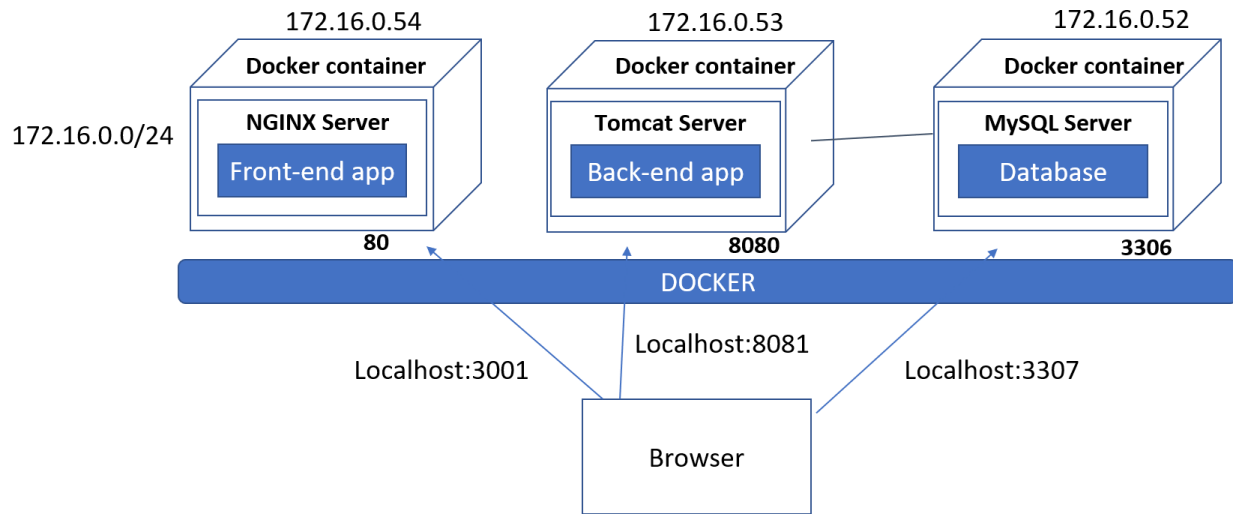


Figure 11 Deployment Architecture

On the host computer runs the docker runtime that will host three containers, one for each application:

- Docker container for frontend application – runs a NGINX server and maps local port 80 to host computer port 3001
- Docker container for backend application – runs a TOMCAT server and maps local port 8080 to host computer port 8081
- Database container for database server – runs a MYSQL server and maps local port 3306 to host computer port 3037

This means that from the host computer we can access the servers within the containers as follows:

- Frontend application: localhost:3001
- Backend application: localhost: 8081
- MySQL server: localhost:3307

These connections allow running and accessing the frontend and the backend application from the browser. However, they do not allow the backend application connect the MySQL server, since **localhost:3306/3307** from the backend docker container would try to connect to the local port 3306/3307 within the container, and not the docker container with MySQL database.

To solve this problem, we need to define a docker network to allow addressing the containers within the network. Thus, containers from a docker network can be accessed similarly to computers in a LAN. In the example provided in Figure 11, the Spring application will access the DB using the address of the container and its port as follows: 172.16.0.52:3306.

Thus, a call to the database through the application will have the following flow (for instance get all students):

- The browser will access the application using localhost:3001, reach the NGINX server running on port 80 on the container, and downloads the React application
- The browser, running the React application, will make the REST call to the backend to the address localhost:8081, that is redirected to the port 8080 of the backend container.
- The Spring application from the backend container will further connect to the database by accessing the MySQL container using the address within the docker network 172.16.0.52:3306 (and not port 3307 that is mapped on the host)

## References

- [1] [https://gitlab.com/ds\\_20201/spring-demo](https://gitlab.com/ds_20201/spring-demo)
- [2] <https://www.baeldung.com/spring-boot-docker-images>
- [3] [https://gitlab.com/ds\\_20201/react-demo](https://gitlab.com/ds_20201/react-demo)
- [4] <https://devcenter.heroku.com/articles/heroku-cli#download-and-install>
- [5] [https://docs.gitlab.com/ee/user/project/deploy\\_tokens/](https://docs.gitlab.com/ee/user/project/deploy_tokens/)
- [6] <https://developer.okta.com/blog/2020/06/24/heroku-docker-react>
- [7] <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- [8] <https://www.computerhope.com/jargon/u/user-space.htm>
- [9] <https://en.wikipedia.org/wiki/Hypervisor>
- [10] <https://devopscon.io/blog/docker/docker-vs-virtual-machine-where-are-the-differences/>
- [11] <https://www.docker.com/blog/vm-or-containers/>
- [12] <https://www.sciencedirect.com/topics/computer-science/hypervisors>
- [13] <https://docs.docker.com/registry/>
- [14] <https://docs.docker.com/develop/develop-images/baseimages/>
- [15] [https://en.wikipedia.org/wiki/Virtual\\_machine](https://en.wikipedia.org/wiki/Virtual_machine)