LAB 4: DOCKER CONTAINERS

The goal of this lab session is to provide you with an introduction on elementary operations to create, start, stop and deploy Docker containers. We will also look into how you can manage multi-container applications on a single host.

1 Setup

Docker is pre-installed on the provided virtual machine. You can login with the following credentials:

- Username: `student`
- Password: `student`

Bear in mind that you need `sudo` rights to execute Docker commands. In the VM, we have already set `sudo` rights for the user `student`.

2 Target deployment

During the course of the next two lab sessions you will containerize the hospital application you developed during the first lab sessions. We will look at how to manage a multi-container application, and scale the number of service instances when demand increases.

In this lab session you will learn about how to start, stop and inspect containers, you will write Dockerfiles and a docker-compose file to allow your micro-service architecture to be deployed with a single command. The containers you will deploy are shown in Figure 1. For the grey blocks in this figure, you can run a container from an image that is available in the Docker Hub. The white color indicates the services for which you will create your own image by writing a Dockerfile.

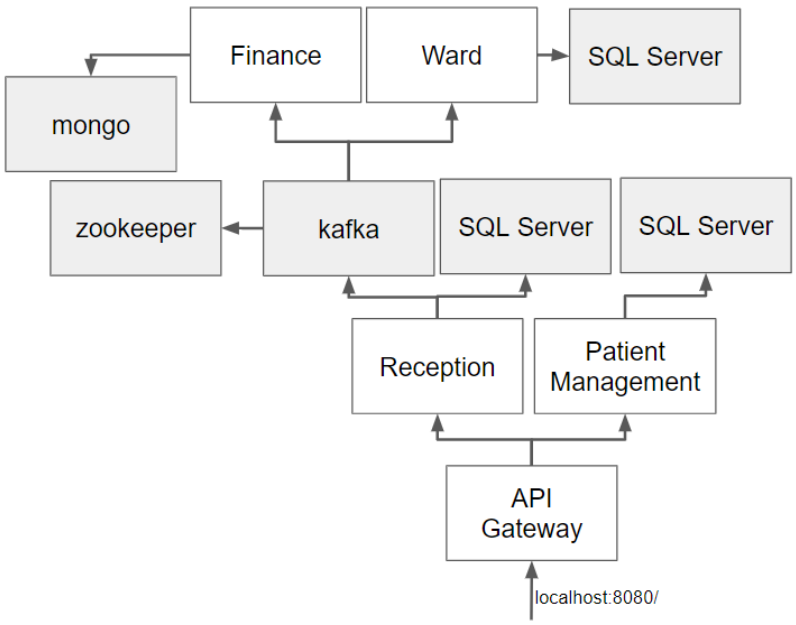

Figure 1: The containers you will deploy during this lab session. Arrows indicate dependencies.

3 Understanding the Docker set-up

You should know about Docker images, containers, etc. from the theory course. For more information on the overall architecture of Docker see: https://docs.docker.com/engine/docker-overview/

Before containerizing we need to look at how to run, start, inspect and stop containers. The easiest container to start is from the Hello-World image. Try to start it.

```
> docker run hello-world
```

This command will pull the hello-world image from the Docker Hub and start it. The Docker Hub contains many more ready-to-use images, we will use some of them later on in this lab.

You should see a welcome message indicating your Docker setup is working.

You will notice your container does not keep running. It just prints a message and then stops. However, the container itself has not been removed yet.

You can check all containers that are defined on the system and their status with the following command:

```
> docker ps -a
```

Inspect the output of this command. You can see the status of all the containers including a unique name given to the container. For more details about a specific container run:

```
> docker inspect <container name>
```

Execute the `hello-world` image again, but give the container a name by yourself, e.g. “my-hello-world”.

```
> docker run --name my-hello-world hello-world
```

After having started this second container, validate your work by checking the output of `docker ps -a`. The unique name of a container can be used in Docker commands to manage that container. For example you can restart your container:

```
> docker start my-hello-world
```

To verify that the container has now been ran twice use the following command:

```
> docker container logs my-hello-world
```

When executing the start command the container ran in the background and your terminal was not attached to the `std_in/std_out`. Start it again but this time attach to it. To find out how, use:

```
> docker start --help
```

Containers can be removed automatically when they stop be adding the `--rm` flag. They can also be removed using `docker rm <name>.` To remove all stopped containers run:

```
> docker container prune
```

There are many more docker commands available, however we won’t go into the details of all of them. See: https://docs.docker.com/engine/reference/commandline/docker/

4 Multi-container Application

In this next section you will deploy a Docker application requiring multiple containers. Think about and discuss the following question:

- Why is deploying multiple containers useful and what applications can benefit from this?

We will start by launching a MySQL container, then create Docker container images for the Patient Management and API Gateway services. We will connect the Patient Management service to the API Gateway and MySQL server. The container architecture for this is depicted below:

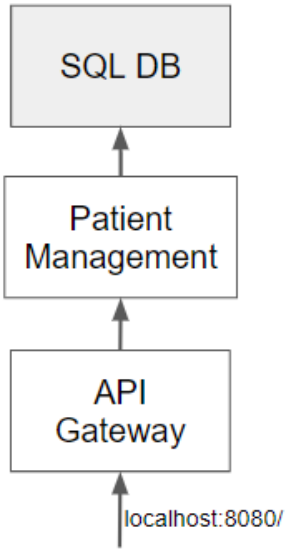

Figure 2: The containers you will deploy during Section 4 of this lab session.

4.1 MySQL container image

We will start by the simplest solution possible: simply pulling an image from the Docker Hub and use it as-is.

- Search the Docker Hub registry for a ready-made MySQL image, via its search interface or by using the command line interface. You will find many results in public repositories from different users. The top-level namespace contains Official Repositories which have been validated and curated. These official repositories should always be your first option to consider!
- Run a container from the chosen image with the following command line flags:
    - a self-chosen name (e.g. patient_db)
    - with the –d flag (why is this flag needed? – you can find the answer in the Docker documentation or in the command line help)
    - two environment variables: 1) to set the *root* user's password and 2) to create a Patient database.

Because the MySQL image contains a daemon as `ENTRYPOINT+CMD`, the container keeps running. Try attaching to your container. Why are you unable to enter any commands?

To open a bash (`/bin/bash`) terminal **inside** your container use the `exec` command. You will need to use the interactive option and allocate a pseudo-TTY.

- What files does the container contain? What commands are available?

- The `entrypoint.sh` file has been set as the `ENTRYPOINT` of the container. The `CMD` instruction, which can supplied to the Docker start command or provided in a Dockerfile, is passed to the `ENTRYPOINT` when the container is started. (see: https://docs.docker.com/engine/reference/builder/#cmd for more details)

4.2 Patient Management container image

We will now prepare an image that contains the Patient Management application. Like in the previous section, we could start from an image in the Docker Hub. But this would require us to add all the Patient Management application's files and call the command to run the application every time you want to start a Patient Management container. Therefore, we will create a new Patient Management image by writing and running a Dockerfile.

- Read the "Define a container with a Dockerfile" part of the Docker user guide: https://docs.docker.com/get-started/part2/#define-a-container-with-a-dockerfile
- The commands that can be used in a Dockerfile can be found on this page: https://docs.docker.com/engine/reference/builder/

While building our Patient Management container, we will explore the use of Dockerfiles.

4.2.1 Your first Docker file

First we will create a Docker container based on the `openjdk:8` image which will run the Patient Application.

In the directory containing your Patient Management application create a Dockerfile which:

- Is based on `openjdk:8`
- That downloads `curl` using the package manager. You will need to update the package manager before installing `curl`. Make sure none of the commands require manual input. Also make sure you use only a single `RUN` directive, because every new `RUN` will create additional layers. You can concatenate bash commands using `&&`.
- Copies your application into the container image. You will need to copy the whole `hospital-application-patient-management` directory, as a compiled JAR file does not include the `application.properties` file.
- Builds the application. The `./mvnw package -DskipTests` will create a runnable JAR file.
- Executes the application. The location of the JAR file should be: `<directory you copied the source files to>/target/hospital-app-patient-management-0.0.1-SNAPSHOT.jar`

Move to the directory containing your Dockerfile and execute:

```
> docker build -t patient . (←notice the dot)
```

Run your container to see if everything works.

Try running: `curl localhost:2222/patient` Why does this fail?

Containers are isolated environments, and by default, when you create a container, it does not publish any of its ports to the outside world. In our application (see Figure 1), only the Reception service should have an exposed port.

To test the Patient Management application:

- run the curl command (`curl 0.0.0.0:2222/patient`) inside your container.
    - See: https://docs.docker.com/engine/reference/commandline/exec/

4.2.2 Connecting to MySQL

At the moment your Patient Management application is using an in memory SQL database, namely h2. If you were to start multiple instances of this service and update a patient's details in one service, the updated information should be available to all instances of the Patient Management container. To make this possible we will use a remote (to the container) SQL server.

For the Spring boot tutorial for connecting to MySQL see: https://spring.io/guides/gs/accessing-data-mysql/

Modify your Patient Management application to use the MySQL server:

- In the `pom.xml` file, replace the h2 dependency with a dependency for `mysql-connector-java`
- Add the required properties to the `application.properties` file.
    - The user name is *root.* Use the password you set when starting the MySQL container.
    - Instead of localhost use the name of your MySQL container.
    - You also need to tell Spring boot which version of SQL you are using:
        - `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect`

Re-build the Patient Management Docker image and check it runs.

Containers provide isolated environments for running code. Therefore, when starting the Patient Management contain we need to link it to the MySQL container.

- Use the `--link` option when starting your container to link it to the MySQL container.

Test your containers are correctly set-up.

- Run the command below to check the Patient's table has been populated.

```
> docker exec –i <name_of_mysql_container> mysql –u root –p –e “SELECT * FROM Patient.patient”
```

- Check that the output of running curl within the Patient container still lists all patients.

If many users attempt to access the website you want to be able to run multiple Patient containers to distribute the load. All instances of your Patient application should be able to access the latest data from the MySQL database. Let's test this out.

- Run a second Patient container from your image, give it a different name and link it to the MySQL container.
- Within one of the containers, add a Patient by executing the following curl command:
    - `curl 0.0.0.0:2222/patient -X PUT -H "Content-Type:application/json" -d "{\"ssn\":\"3\",\"firstName\":\"Tom\",\"lastName\":\"Palmer\",\"dateOfBirth\":\"2001-07-25\"}"`
- Inside your other Patient container use the curl command to check that it can access the newly inserted data.

4.3 API Gateway Container image

Like the Patient Management application, we need to develop a Dockerfile for the API Gateway application. Instead of copying the application files into the image, we will use a bind-mount (https://docs.docker.com/storage/bind-mounts/). When you use a bind mount, a file or directory on the host machine is mounted into a container.

Rather than rebuilding an image every time you make a change, a bind-mount enables any changes to the application on the host machine to be visible from within the container.

- You could develop applications directly within containers. What are the disadvantages of modifying documents directly on containers?

- What are the disadvantages and advantages of mounting a directory?

Before creating the image, the API Gateway application code should be modified to point at the Patient service.

- In the class containing the main method, modify the URL. Rather than “localhost”, you should use the name of the Patient container.

Create a runnable JAR.

- In a terminal window navigate into the directory for the API gateway application and run: `./mvnw package -DskipTests`

Write a Dockerfile for the API Gateway Service.

- Start from the `openjdk:8` image
- Include the command to run your application (don't copy across the application)

Build and run your API Gateway image. When starting the container you should include the following options:

- Provide a name for the container
- Link it to the Patient container
- Use the volume flag to mount the directory containing the application to the container.
  - See: https://docs.docker.com/storage/bind-mounts
  - You need to use absolute paths. To get the current working directory on the host machine you can use: "`$(pwd)`".

You can't use the web service from your host machine yet since the ports have not been exposed. There are two ways to expose the port, you can select the port mapping manually with the `-p` option or you can do it dynamically using `--expose` and `-P`. The documentation is available at: https://docs.docker.com/engine/reference/run/#expose-incoming-ports

We would like to access the reception application by visiting `http://localhost:80/patient`.

- Adapt your run command to achieve this.

4.4 Docker compose

Docker compose enables developers to start multiple Docker containers using a single command: `docker-compose up`. For more information and some use cases see: https://docs.docker.com/compose/overview/.

Create a Docker compose file which allows you to start-up our hospital application. The guidelines for Docker compose can be found at: https://docs.docker.com/compose/compose-file/. We are using Docker compose version 2. You should think about:

- What order the containers need to be started in
- What options are required to run the different containers
- How to build the Patient and API Gateway images when the docker-compose file is ran

Run your docker compose file. If correctly set-up you should be able to list all patients by visiting http://localhost:80/patient.

5 Reception, Ward and Finance containers

In the same way we have containerised the Patient and API Gateway applications, you should now containerize the Reception, Ward and Finance applications. You can choose to either copy the application files across or use the `volumes` option.

- Write a Dockerfile for the Reception, Ward and Finance applications
- Modify the Docker compose file to run a Reception, Ward and Finance container.

The Finance application stores data in MongoDB, therefore we also need to start a container running Mongo.

- Add a Mongo container to the Docker compose file.
  - Use the official mongo image
- Link the Finance service to the Mongo container.
  - You will also need to tell the Finance spring application the host name for accessing mongo: `spring.data.mongodb.host=mongo`

The Reception and Ward services use SQL, thus you need to modify these services to use MySQL and add additional MySQL services to your docker-compose file.

As these three services communicate via a Kafka broker, we also need to run Kafka and Zookeeper containers. Add the following service images to your docker-compose file (you can find more information about the images by search for them on docker hub):

- confluentinc/cp-zookeeper with the following environment variable:
  - `ZOOKEEPER_CLIENT_PORT: 2181`
- confluentinc/cp-kafka which is linked to zookeeper and has the following environment variables:
  - `KAFKA_BROKER_ID: 1`
  - `KAFKA_ZOOKEEPER_CONNECT: <zookeeper service name>:2181`
  - `KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://<kafka service name>:9092`
  - `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1`

You need to tell your spring applications the host name for Kafka and Zookeeper.

- Add the following properties to the relevant `application.properties` files:
  - `spring.cloud.stream.kafka.binder.brokers=kafka`
  - `spring.cloud.stream.kafka.binder.zkNodes=zookeeper`

Run your docker-compose file. Try visiting:
http://localhost/reception/check_in_patient?patientId=1