Introduction to Cloud Computing – Exercise 1

Scope: The lab is devoted to preparing the runtime environment to work with Docker. The first task is to install Docker. The next task is to run the sample web application.

1. Preparing the environment and downloading the repository

Make sure you have Ubuntu 20.04 installed on your computer or virtual machine before proceeding. All commands must be entered in the terminal.

For proper installation, make sure no other version of Docker is installed

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

Other versions should be uninstalled. In case there was no Docker installation, *apt-get* will report that no action has been taken.

There are different methods to install Docker, but in this lab we will focus on repository-based installation.

Update the apt indexes and install the necessary components to be able to download the repository via HTTPS

```
$ sudo apt-get update
$ sudo apt-get install \ ca-certificates \ curl \ gnupg \ lsb-release
```

Then add the official GPG key

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

After executing the command, we can select the repository to download.

```
$ echo \ "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
```

```
https://download.docker.com/linux/ubuntu \ $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

2. Docker Engine installation

Re-update the *apt* indexes and install the latest version of Docker Engine and *containerd*

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Verify the correctness of the installation by executing the *docker run* command by running the *hello-world* test image

```
$ sudo docker run hello-world
```

3. Docker run

After proper installation, we can proceed to familiarize ourselves with running images on our Docker. The first step will be to download and run the BusyBox container as an example.

To get started, we first download the image

```
$ docker pull busybox
```

If an error pops up when you try to download, it is related to the fact that we are trying to execute the command without administrator rights. At this point, we have two options: we can execute this command by adding the *sudo* prefix or call the *sudo -s* command and then all commands will be invoked with administrator rights.

Correctly performed command downloads the Busybox image from the docker repository and saves it in our system. Then we can view all downloaded images

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
busybox             latest              c51f86c28340        4 weeks ago         1.109 MB
```

Once we have the image downloaded, we can start a new container

```
 $ docker run busybox
```

In the console we will notice that nothing else happened. This behavior is correct. By executing the *docker run busybox* command, three actions took place: 1. The Docker client found an image

with the specified name, 2. A new container was created, 3. The command was executed on the given container. When calling *docker run busybox*, we do not specify any command to run on the container, so the container started and then closed. Execute *docker run* by specifying the command

```
$ docker run busybox echo "hello from busybox"
```

What is the output? Why?

4. Docker ps

In the previous section, we launched a container containing the Busybox image. The docker ps command is used to list all active containers. Use the command

```
$ docker ps
```

As there are currently no active containers, the list is empty. For more information, use the command

```
$ docker ps –a
```

What is the result of the given command? Explain the relevant columns.

Let's try to run more than one command while running the container with the busybox image

```
$ docker run -it busybox sh
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # uptime
 05:45:21 up  5:58,  0 users,  load average: 0.00, 0.01, 0.04
```

Starting a new container with the *–it* flag connects ours to the interactive *tty* interface and gives us the ability to execute commands “inside” the container. Use a few commands to test running them in a container. To exit the container, use the exit command.

The docker run command is the most common command when working with docker. In order to continue working with docker, it's worth taking a moment to get to know it better. To see all possible flags for the docker run command, run it with the *–help* parameter

```
docker run --help
```

5. Docker rm

After practice with running containers, the next step is to familiarize yourself with the container removal mechanism. Although all containers have finished their work, you can display them with the *docker ps -a* command. Non-working containers take up space on the disk, to remove them, run the *docker rm* command and the container ID, e.g.

```
$ docker rm 305297d7a235 ff0a5c3750b9
305297d7a235
ff0a5c3750b9
```

You should see an ID return as confirmation of the container removal. Another method of deletion is to indicate the deletion of all containers that have finished their work.

```
$ docker rm $(docker ps -a -q -f status=exited)
```

The -q flag returns the IDs of the containers and the -f flag filters the results so that the status field has a specific value. The last thing worth mentioning is the -rm flag when invoking the docker run command, which automatically deletes the container when it's finished..

6. WebAPP

After learning the basic commands, the next task is to run the web application on Docker. The first step is to display a static website hosted on a Docker server. To do this, download and run the sample website

```
$ docker run --rm prakhar1989/static-site
```

Since the image does not exist locally, the first thing to do is download the image and then run the container. If everything goes well, you should see Nginx is running… Just starting it does not configure the container: it has no address or port. To quit the container, press Ctrl+C. To be able to run the container with parameters, execute the command

```
$ docker run -d -P --name static-site prakhar1989/static-site
```

In this case, we used the -d flag to unbind the terminal from our container, -P to assign random ports to the container, and -name to give a specific name to our container. To see which ports have been assigned to our container, execute the command

```
$ docker port static-site
```

Then we should get a list of ports assigned to our container.

Go to http://localhost:PORT_NUMBER (use mapping to 80/tcp).

The page should appear.

7. Docker images

We've run downloaded images in the previous sections, and in this section we'll explore finding and creating your own images. To list all local images, execute the command

```
$ docker images
```

You should get a list of downloaded images in the console. Each of them has three fields:

Repository – this is the name and origin of the image. In case the image is official, it only has a name (e.g. python, busybox) if the image is created by a user, it has its name first and then the name of the image.

Tag – this is the description of the version of the image. It can simply be the latest (latest) or, as in the case of specific software, have its own version (e.g. in the case of ubuntu, you can specify the version: ubuntu:18.04)

Image ID – this is a unique tag of a given image.

If we wanted to search for an image. We can use the docker search command for this purpose.

The next task is to create our own image and load it into the container. The first command is to download the prepared repository with the web application.

```
$ git clone https://github.com/prakhar1989/docker-curriculum.git
$ cd docker-curriculum/flask-app
```

The downloaded application randomly downloads and displays .gif files with cats. This is our target application that we want to run. The next step is to build your own image. The application is written in Python, so our base image will be Python3. To create our image, we will prepare a Dockerfile. This is a collection of commands that are invoked during image creation. This way we can automate the image creation process. It should be added here that the commands used in Dockerfiles are practically identical to Linux commands - therefore there is no need to learn another language.

Make sure you are in the docker-curriculum/flask-app folder. Create a new file in this folder (you can use a text editor), it must be in the current folder and save the file as Dockerfile. Then open the file and add the command

```
FROM python:3
```

This command specifies the base image. Then we copy the files and dependencies.

```
# set a directory for the app
WORKDIR /usr/src/app

# copy all the files to the container
COPY . .
```

Now we can install what we copied

```
# install dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

Next, we will specify the port on which our application inside the container should run

```
EXPOSE 5000
```

The last thing is to launch the application

```
CMD ["python", "./app.py"]
```

The main task of the CMD command is to tell the container what command it should execute at startup. This is all that should be in the Dockerfile. At this point, we can build our image. To do this, run the docker build command. You need a Docker hub account to build. If you don't have one, you should create it first (you can use your university email). After creating your account, execute the command replacing **yourusername** with your own username

```
$ docker build -t yourusername/catnip .
```

If everything went well, you should see a message that the image was built successfully. Then you can run our built image.

```
$ docker run -p 8888:5000 yourusername/catnip
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

After entering localhost:8888, we should see a website. Present the result to the teacher.